UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

EVENT PROCESSING AND CORRELATION FOR A CYBER DEFENCE CENTER

Pedro Miguel Rainho Chaves

MESTRADO EM SEGURANÇA INFORMÁTICA

Dissertação orientada por:
Prof. Doutor António Casimiro Costa
e co-orientada pelo Mestre Luís Miguel Costa

2017

Acknowledgments

I would like to express my sincere gratitude to my thesis advisor Luís Costa who always gave me the freedom to make this work my own while motivating my research and steering me in the right direction. I would also like to thank Prof. Antonio Casimiro who was always available whenever I had a question or a problem about my research or writing. I am also thankful for all the expert help provided by Ricardo Oliveira during the development of this project.

I am deeply grateful for all the continuous support and motivation offered by all members of the Siemens CDC team. A special thank you to André and Rodrigo who were responsible for the setup of the development environment and who also helped me to overcome some initial obstacles.

I would also like to thank my colleagues Luís, Sérgio and Raimundo for all the motivation they gave me and all the memorable moments we had during our masters degree.

Lastly, I must express my profound gratitude to my parents, brother, sister and girlfriend for providing me with unfailing help and encouragement throughout all my years of study and without whom this accomplishment would not have been possible.

Resumo

O rápido avanço tecnológico que se tem vindo a verificar ao longo dos últimos anos, transformou drasticamente a forma como as organizações gerem o seu negócio. Os sistemas computacionais permitem um rápido processamento dos dados, facilitam a recolha e armazenamento de informação e, em alguns casos, possibilitam a automatização de processos recorrentes. Adicionalmente, o surgimento e evolução da Internet aumentou exponencialmente a troca de informação, facilitando a colaboração interna e entre diferentes organizações. Este novo paradigma, permite um maior foco na análise da informação produzida, levando a um aumento tanto em produtividade como eficiência.

Para que a informação possa circular de forma segura, é necessário que todos os sistemas e padrões de comunicação sejam monitorizados. No que diz respeito à segurança da informação, esta monitorização é essencial especialmente quando considerando que os componentes de uma organização estão geralmente interconectados, implicando que potencial *software* malicioso possa disseminar-se com facilidade, tendo consequências desastrosas. Mesmo na ausência de ataques externos, os utilizadores poderão lançar ataques internos ou difundir informação confidencial.

Uma das formas de identificar potenciais intrusões e comportamentos indevidos é através da coleção de dados de registo gerados pelos sistemas. Os dados de registo contêm informação crucial relativamente à infraestrutura de uma organização. Por exemplo, os registos de computadores pessoais fornecem informação sobre potenciais infeções ou comportamentos suspeitos, os registos de autenticação permitem acompanhar o movimento dos acessos às contas dos utilizadores pela rede e os registos de servidores *proxy* permitem identificar acessos a sitos web potencialmente perigosos.

A recolha de dados de registo é apenas o primeiro passo para alcançar o objetivo principal. Isto é, inferir significado a partir dados colecionados. Sem as técnicas apropriadas, encontrar introspeções acerca dados recolhidos pode ser como ”tentar encontrar uma agulha num palheiro”. Os principais desafios incluem normalização, armazenamento e análise de informação, o que suscita vários problemas. Em primeiro lugar, diferentes sistemas podem armazenar os registos em diferentes formatos. Em alguns casos, a informação relevante poderá estar entre dezenas de campos pobremente estruturados. Em segundo lugar, o número total de eventos poderá ser colossal. O aumento da tecnologia por sensores (e.g. leitores de cartões) e a necessidade de monitorizar os sistemas levou

a um aumento significativo na criação de dados de registo. Estes conjuntos poderão ser tão grandes e complexos que não podem ser armazenados e analisados de forma eficiente em bases de dados tracionais. Por último, é necessário correlacionar toda a informação obtida a partir de diferentes fontes de dados de modo a detetar anomalias.

Atualmente, existem vários sistemas cujo objetivo é lidar com esta problemática. Algumas das soluções propostas representam sistemas de ficheiros distribuídos onde a informação é armazenada num formato que permite uma análise simples e rápida. Geralmente, estão associados a estes sistemas ferramentas que suportam consultas sobre dados armazenados. Para esta análise, tipicamente é colecionado um conjunto relativamente grande de dados e seguidamente são feitas consultas sobre estes, possibilitando, por exemplo, saber quantas vezes um utilizador acedeu a um determinado servidor durante um determinado período de tempo. Apesar deste método poder fornecer várias introspeções importantes sobre a informação recolhida, não permite uma resposta imediata pois os dados são primeiro armazenados antes da análise, o que introduz uma latência significativa. Em arquiteturas modernas existe a preocupação de processar os dados com baixa latência para que, se uma determinada condição de verificar (e.g. intrusão num sistema), seja despoletada uma ação imediata. Para além disso, os dados a serem analisados não são finitos pois estão continuamente a ser gerados, o que por sua vez agrava o problema.

De modo a lidar com o fluxo continuo de dados e evitar a sobrecarga associada ao armazenamento, foram propostos vários sistemas de Processamento de Fluxos (do inglês *Stream Processing engines*). Estas soluções são capazes de processar grandes quantidades de dados mantendo latências reduzidas. Contudo, estas geralmente não fornecem meios que facilitem uma análise mais complexa que pode incluir, construções temporais, correlação de dados de diferentes fontes e deteção de sequências. Para este fim, normalmente são utilizadas técnicas de processamento complexo de dados (do inglês *complex event processing*). Estes sistemas foram desenhados especificamente para lidar com fluxos contínuos de dados, permitindo uma análise complexa dos mesmos e uma resposta imediata no caso de se verificarem determinadas condições. Podendo assim evitar a propagação de um ataque sofisticado.

Apesar de úteis, os sistemas de processamento complexo de dados existentes, não fornecem módulos de análise por omissão, apenas meios que possibilitam essa construção. Para além disso, muitas das funcionalidades e garantias fornecidas não são as desejadas. Como tal, o objetivo do presente trabalho é fornecer uma solução completa que permita uma análise complexa de eventos numa perspetiva de segurança, fornecendo também meios para colecionar, agregar e analisar os eventos com facilidade, mantendo a disponibilidade do sistema e evitando a perda de dados, utilizando como base algumas soluções já existentes.

Palavras-chave: Correlação de Eventos, Apache Flink, Processamento de informação, Segurança da Informação, Monitorização

Abstract

The growth of computer systems in information technology infrastructures accompanied by an increase in their interconnectivity implies not only that critical information can now be scattered across different systems, but also that an intrusion can easily spread and have disastrous consequences. From a security standpoint, this means that every computer system must be actively monitored and if a suspicious activity is detected, a quick countermeasure must be taken. Many intrusion detection systems and prevention mechanisms struggle to monitor all the components, both because the amount of information generated is too large and complex to be interpreted, and also due to the sophistication of current attacks. Moreover, a possible attack must be quickly identified in order to avoid further propagation and damage. In this work, we present a solution resorting to existing frameworks that is capable of correlating information from different sources in order to quickly identify possible security breaches from continuous streams of events.

Keywords: Event Stream Processing, Complex Event Processing, Event Correlation, Apache Flink, Information Security

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The evolution of technology has drastically changed the way organizations conduct their business by allowing the automation of tasks and simplifying the storage, retrieval and analysis of information. Predominantly, the development and widespread of the Internet has exponentially increased the rate at which information is exchanged, empowering cooperation internally and between organizations. Unsurprisingly, this new shift in technology enforces the need to monitor Information Technology (IT) systems, for instance, due to the proprietary information that these might contain.

From a security standpoint, it is important to monitor every component in an infrastructure and to keep track of communication patterns to maintain a secure flow of information. This motorization is essential especially when considering that systems are generally interconnected, consequently, malicious software can easily spread and have disastrous consequences. Even in the absence of external threats, users may potentially launch attacks from within the organization or leak important information.

One way to keep track of potential intrusions and misconducts is through log[1] collection. For instance, event logs from individual computers provide information about potential infections and misbehaviors, authentication logs are used to keep track of user accounts moving throughout the network and proxy logs can provide insights of potentially dangerous accesses to external websites. For security purposes, a log is used to record data on who, what, when, where and why (W5) an event occurred for a device or application [1].

Log retrieval is only the first step to archive the main goal, which is to infer significance from the gathered data. Without the proper techniques, finding insights can be like searching for a needle in a haystack. Challenges include parsing, storage and analytics, which raises several problems. First, different systems contain logs in different formats and, in some cases, the relevant information is amidst dozens of unstructured fields. Second, the number of events per second can be tremendous. Sensor technology (e.g. card readers) and the need to monitor systems has led to a significant increase in log genera-

[1] A log can be seen as a record of the operations performed by a machine.

tion. These data sets are so large and complex that they cannot be efficiently stored and queried on traditional databases. There are already several solutions that deal with this problematic [2, 3, 4, 5]. These proposed solutions are based on a distributed file system, which allows them to easily scale in order to store large amounts of data. Although we can typically query these file systems, the latency associated with storing and retrieving the data does not allow for an immediate response. In modern infrastructures, there is a need for data processing with very low latencies such that if a condition is met an immediate action can be taken.

To deal with the continuous flow of data and avoid the storage overhead a different approach known as Stream Processing was proposed. This method provides means to process massive amounts of data with low latency by keeping events in memory and processing them sequentially, such that if a certain condition is verified a quick action can be taken. To complement stream processing, a new technique named complex event processing (CEP) was proposed. CEP provides mechanisms to detect complex patterns by analyzing relationships between events. Both techniques are discussed in Chapter 2.

The existing event stream processing engines do not usually provide all the required features out of the box. Besides the installation, configuration and tunning, it is often necessary to program the desired behavior to fulfill a particular need, which implies that some new functionality or guarantees must be provided. The present work builds upon already existing frameworks to provide such a system, which in this case is specifically designed and optimized to deal with security-related events to quickly identify possible security breaches.

1.1 Motivation

The presented dissertation emerged from a partnership between Faculdade de Ciências da Universidade de Lisboa and the company Siemens.

Siemens is a German company that has ongoing operations in more than 200 countries around the globe. As of September 30, 2016, Siemens had about 351,000 employees and generated revenues of 79,6 billion dollars. The company focuses on areas of electrification, automation and digitalization offering a wide range of products and services which include power generation services, energy management, mobility, manufacturing, process industries, financial services and health-care.

Due to its large customer base, number of employees, widespread and criticality of the services, Siemens actively monitors its infrastructure and activities. To monitor this infrastructure Siemens has a Cyber Defence Center (CDC) in Portugal that shares activities with Munich (Germany) and Milford (USA). The CDC team is responsible for analyzing data derived from the systems and identifying security threats.

Because of the thousands of systems that Siemens has spread all over the world, monitoring all the components is not an easy task, especially because there is a need to quickly process and react to them when certain conditions are met. The current solutions raise several problems that prevent Siemens monitoring capabilities to grow beyond the current scope. Commercial solutions tend to be very expensive due to licensing prices, technical support costs and the requirements of high-end servers because there is no support for horizontal scaling (i.e. the ability to distribute the computations across several machines). Moreover, the provided software is generally easy to use but at the cost of losing some flexibility in the development of security rules. On the other hand, open source solutions, although more flexible, lack some of the required features or are not able to handle the amount of generated events. As such, a new solution is required.

1.2 Objectives

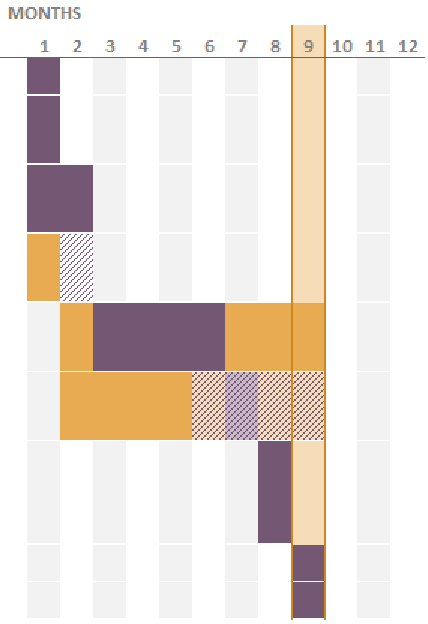
Faced with the challenges at Siemens our task is to present a solution that is able to address them providing means to collect, normalize, aggregate and analyze log data. To accomplish these goals, we subdivided our work into several objectives. Primarily, we need to establish some groundwork by getting familiarized with the concepts related to stream processing and by studying the current works and solutions. Secondly, we need to identify which of the studied solutions could be used to develop the system and pick the one that best fits our needs. The third objective includes the most fundamental part of the work, which is the design and implementation the system using the previously selected engine. Lastly, we need to test, evaluate and incorporate the application in the production environment.

1.3 Planning

This section describes the project plan in order to accomplish the established objectives. The initial work plan is presented as a Gantt Chart in Figure 1.1. Following the general plan is a description of each task.

1. **Initial Investigation:** Begin with a study of the current Siemens architecture followed by an investigation of the existing event processing frameworks as well as a comparative study between those frameworks.
2. **Create a Proof of Concept Environment:** Setup a virtual machine and install the most relevant engines identified in the investigation phase for demonstration proposes.
3. **Crete a demonstration application:** Implement a small application that allows the comparison of the frameworks in terms of ease of development, configuration, extensibility and integration with other tools.

ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Investigation	1	1	1	1	100%
Create a POC Environment	1	1	1	1	100%
Perliminary Report	1	2	1	2	100%
Create a Demo Application	2	1	1	1	100%
Design and Implementation	3	4	2	8	100%
Fault Tolerance Testing	7	1	2	8	50%
Capacity and Scalability Planning	8	1	8	1	90%
Integration	9	1	9	1	100%
Final Report	9	1	9	1	100%



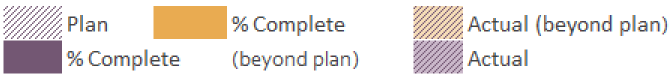


Figure 1.1: Project plan

4. **Design and Implementation**: First, setup a realistic source of log data and create some sample security rules in order to test the correctness of the application. Create a module that allows users to dynamically add, remove or edit rules. Develop the necessary functionality to allow a complex analysis of event data and the extraction relevant information. Finally, handle out of order and late events.

5. **Fault Tolerance Testing:** Simulate network delays, component failures and stress test the application.

6. **Capacity Planning and Scalability:** Plan the necessary hardware to handle the current number of events and make a scalability plan.

7. **Integration:** Integrate the solution with the existing architecture.

Chapter 2

Background

A comprehensive study of the concepts related to event processing is essential not only to pave the groundwork for the proposed solution but also to provide a better understanding of the reasoning behind the design decisions. As such, this chapter describes the mechanisms of both batch and stream processing while highlighting some of the underlying challenges. In Section 2.1 we begin with a description of the concepts related to Event Processing and the techniques that can be used to perform event analysis. Section 2.2 highlights some applications of the described techniques and outlines an architectural model that includes an event processing engine. Section 2.3 describes a programming model that many languages have adopted over the last years. This model creates an abstraction in which a set of chained operations are used to process data and whose ideas have been adopted by the majority of event processing engines. In Section 2.4 we compare two types of architectures that are commonly used for Event Processing - Lambda and Kappa. In Section 2.5 we describe two types of memory-efficient data structures that can be used to represent large sets of elements whilst maintaining a low memory occupation. Potentially holding all the necessary information in memory is a great plus because we avoid the latency associated with disk reads and writes. Lastly, in Section 2.6 we provide a description of some information security concepts, which are referenced throughout the work.

2.1 Event Processing

The term Event Processing refers to techniques which aim to extract knowledge (i.e. relevant information) from events. An event can be defined as a record of a notable action performed by a system as a reaction to another event or triggered by a user. The event should contain enough data to allow users or systems to identify what happened and when. Event processing is becoming increasingly more important both because of the rise of Event-Driven Architectures (EDA) [6] that promote the production, detection, consumption and reaction to events, and also due to the possible opportunities or threats that may be identified by analyzing them. For our particular use case, an event record represents a log line that has been collected and, if necessary, parsed to a format that can be easily interpreted. For instance, the Figure 2.1 illustrates a sample firewall log in the Syslog [7] format parsed to the CEF [8] format where the retrieved values are more easily identifiable since they get associated with a particular key (e.g src - source address).

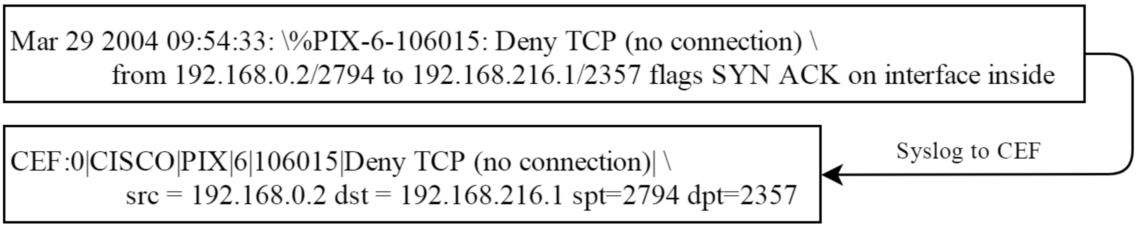
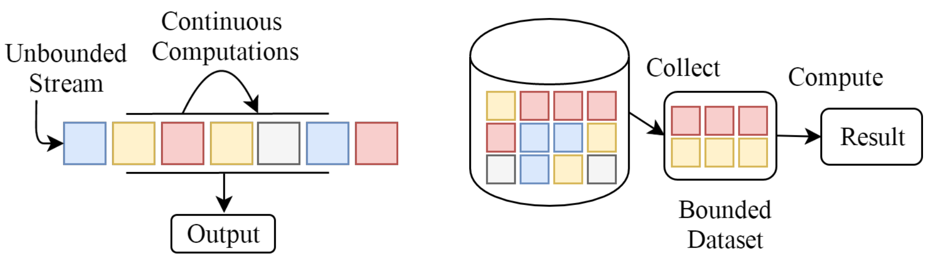


Figure 2.1: Syslog message parsed to the CEF format

Event Stream Processing (ESP) is a type of event processing used to drive a real-time analysis of events. The aim of ESP is to shorten the time between the occurrence of an event and the process of decision-making, that is the action to take in response to the event. This is important because an event is at the peak of its value upon creation. From a business perspective, the quicker an opportunity is identified the quicker systems can react to it in order to increase profits. From a security standpoint, a threat must be quickly identified in order to stop an eventual breach. To perform this analysis, ESP systems must handle event streams, which can be perceived as unbounded data sets. Using the definition from [9] an unbounded data set is a type of ever-growing, essentially infinite data set. As events enter the system, ESP deals with the task of processing them by continuously running computations as depicted in Figure 2.2a. The computations are expressed by queries and represent the logic that must be applied to the stream of events. The queries are usually created in one of two ways: by developers when programming the system or on demand by end users. The aim is to gather the relevant information from the events as fast as possible. For instance, having a stream of stock market values, users may want to query the events for stock values that rise above a certain threshold so that they may sell shares for a quick profit.



(a) Stream Processing (b) Batch Processing

Event Batch Processing (EBP), contrarily to ESP, deals with bounded data sets. Batch processing systems typically apply queries to a very large set of stored events. Before applying the computations, batch systems first collect a set of data, which is typically re-organized to facilitate the followed computations as depicted in Figure 2.2b. Batch queries are usually more complex and can evolve gathering information from several events, possibly from different systems. For example, one could use batch processing on a collection of sales events produced in a whole day in order to infer what products were sold the most, what was the profit and other relevant metrics.

Although batch and stream processing are usually perceived as disparate realities, stream processing can be seen as a super-set of batch processing because stream applications can create bounded data sets from unbounded data using windows, while the opposite is not true.

2.1.1 Windowed Constructs

One simple way we can process streaming data is through simple filtering, that is looking at each incoming event and filter the ones that match a certain condition. For instance, one could maintain a list of known vulnerable ports and match the incoming traffic destination port against the elements on the list in order to infer an anomaly. Although this type of analysis can be useful, it does not satisfy all the required use cases. To name a few, brute force and denial of service attacks require some type of aggregation function (e.g. Count, Sum or average). In order to apply these functions, events need to be grouped into windows forming bounded data sets. Windows are normally [10] based on one of the three following characteristics:

- **Event Time**: Based on the time at which events occur.
- **Processing Time**: Based on the system internal clock.
- **Event Count**: Based on the number of events.

The first two types can be used to form temporal windows, for instance, grouping all the events from the past 10 minutes. The third type is used to create dimension-based windows, where a computation is performed after a certain threshold is achieved. Grouping elements using windowed constructs allows for a new type of logic to be applied. For example, one could apply a function to all elements in a window counting how many times a user has accessed a server in a certain time interval in order to infer a brute force attack. Regardless of the event property which they are formed, windows are subdivided into three types: Tumbling, Sliding and Session windows whose semantics are illustrated in Figure 2.3.

- **Tumbling Windows**: Tumbling or Fixed windows aggregate events into fixed-sized segments. For instance, if a 10 second window is created the function will gather

10 seconds worth of elements before performing the computations. After the computations are performed, all the elements in the window are evicted to make room for another batch.

- **Sliding Windows**: Sliding windows are defined by a window length plus a sliding interval allowing windows to overlap. For example, one could create a window with a 5 minute length and a 1 minute slide. This way the window is invoked every minute. When the window is full, instead of evicting all the elements, it slides 1 minute, which means that the elements collected in the first minute are removed allowing the buffer to aggregate another minutes worth of events.

- **Session Windows**: Session windows are dynamic because contrary to tumbling and sliding windows, they do not have fixed parameters. Instead, they automatically adjust their boundaries according to the incoming data. In this case, windows will be created in periods of greater activity and closed when an interval of inactivity is observed. Session windows are relevant in cases where a length cannot be defined a priori due to signals suffering from jitter or to group together a series of temporally related events. As an example, lets suppose that we are listening to an album and we would like to perform a computation at the end of every song. Using a fixed window will not suffice since the length of every song in the album can vary. In this case, one could use a session window where the periods of inactivity would represent the transitions between each song.

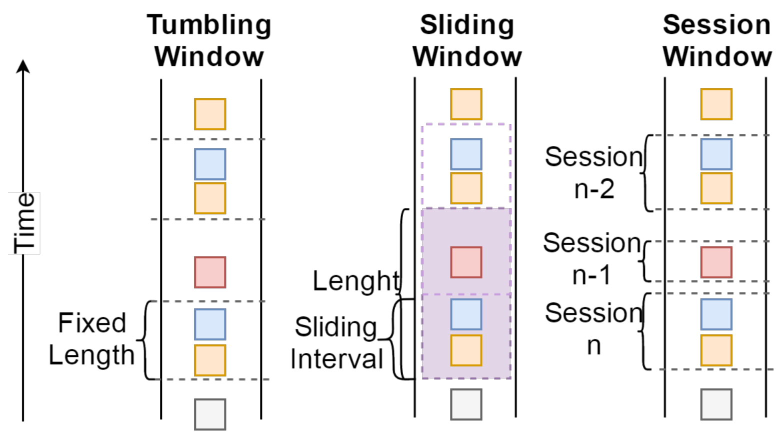


Figure 2.3: Windowed Constructs

2.1.2 Complex Event Processing

Similarly to ESP, Complex Event Processing (CEP) [11, 12] is a method of analyzing streams of events with the added ability to query streams for so called complex events. Complex events manifest themselves upon the occurrence of a defined sequence called a complex pattern and/or by combining events from multiple sources during a certain time window. To illustrate, let us assume a temperature sensor that produces an event stream of temperatures changes over time. Using a CEP engine, users could query the stream for three consecutive temperature increases in a time interval of 5 seconds which may indicate an abnormal temperature rise. These patterns are also useful to represent security rules, which describe the necessary conditions for an attack to happen. As a rule of thumb, CEP engines operate with little to no storage requirements and its queries are designed to respond to the changing conditions/events with very low latency. In addition, since it needs to detect sequences, the events must be ordered by time of occurrence before the rules are applied.

In order to quickly react to events entering the system, CEP engines rely on several techniques to correlate events. Event correlation can be decomposed into several steps which includes Normalization, Filtering, Aggregation, Detecting Relationships and Analytics.

Normalization: To ease the correlation of events (i.e. combining data from multiple sources), these need to be parsed to a pre-determined format in a process named Normalization. Normalization makes possible to correlate similar fields that could be initially in different formats (e.g. Textual and Binary). At a first glance, Normalization can seem simple, but the reality is quite the opposite. Primarily because different systems can have logs in very distinct formats. Furthermore, the components of an infrastructure are constantly changing, a simple update can mean a change in the log format, which implies that the parser for that component must also change. When an architecture has thousands of components this task can be very time consuming and thus expensive.

Filtering: Since some events are of interest while others are not, an initial filtering must be carried out to discard irrelevant events. This filtering is usually a simple match between a set of conditions and the event fields.

Aggregation: Event aggregation is a technique to combine multiple events of interest in a set. The objective is to reduce the stream of events into smaller collections before the analysis. This aggregation can be done by specifying key-values to group events with equal keys and/or by using windows.

Detecting relationships: By knowing which type of event a log refers to can ease the process of analysis since different types of queries can be applied to different types of events. When analyzing web logs, the URLs provide many important insights while for database logs the same might not be true.

Analysis: The analysis is the last and most complex stage which involves all the logic to be applied to the events. The logic is contained in the queries, which may include aggregation functions. Aggregation functions are mathematical functions that are applied to a set of events, for instance, an average or sum. The results of a CEP analysis are interpreted as a new (complex) event that is then sent to other systems for further analysis or to trigger a new action.

In our temperature example, if we had to analyze a stream of several sensors and we wanted to know if the temperature increased more than 5 % in the last 5 minutes one could specify a CEP query similar to the one below.

```
Select event.source, event.data
where event.source == "Temperature"
Trigger Alert if avg(data) increased 5% within 5 min
```

Listing 2.1: CEP query example.

Using the techniques described the events could be submitted through a pipeline for analysis as depicted in Figure 2.4, where each node would process part of the query.

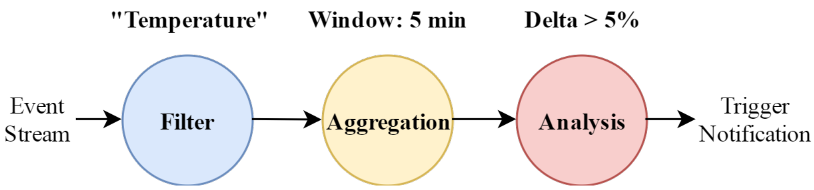


Figure 2.4: Sample processing pipeline

2.2 Stream Processing Platforms

We refer to Stream Processing Platforms (SPPs) as a domain of applications which aim to collect and extract new knowledge from information produced by multiple sources. At its core, an SPP can use an ESP or more specifically a CEP engine to drive the analysis. SPPs are used in a variety of applications that include but are not limited to, financial applications to detect new trends, monitoring credit card transactions, environmental monitoring applications or applications that detect possible security breaches.

SPP architectures are composed of several layers as depicted in Figure 2.5. Each layer has a specific role in the architecture and contributes to the main goal: to derive important conclusions from a stream of continuously produced data.

Event sources are the components being monitored, which regularly produce logs of interest that need to be collected. Event collectors can be seen as observers that gather logs from the sources using one of two strategies: Push or Pull. In the Push strategy log sources are responsible for regularly sending the produced logs to the collectors. In a pull strategy, the collectors are responsible for querying the sources for changes. As

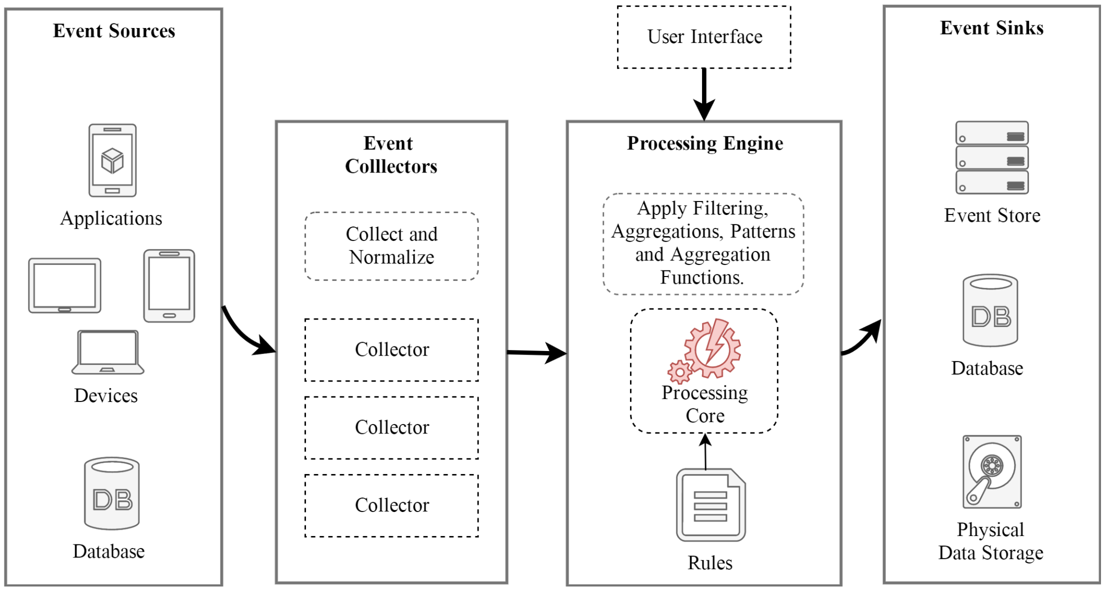


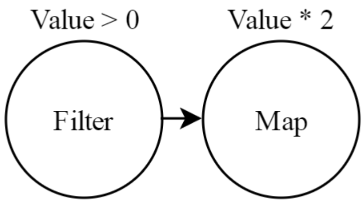
Figure 2.5: SPP Architecture

new logs are observed, the collectors actively retrieve them. Usually, these collectors will normalize the logs to a known format before sending them to the processing engine.

The processing engine is the brain of the solution. Using the techniques described in the previous sections, it will analyze the stream of incoming events. The analysis will directly depend on the specifications given by the processing rules or queries. The rules are submitted by end users possibly using an Event Processing Language (EPL), which is a domain specific language for processing events, or a web interface. These specifications or modules contain all the conditions (e.g. Filters and Aggregation functions) that should be matched against the events to obtain the results. Ultimately, the results of the analysis are sent to the data sinks. Sinks are third party software that may store events, trigger new actions as a response or perform further analysis. The results can also be resubmitted to the Engine as a new event to be correlated with new incoming events.

2.3 Dataflow Programming

The Dataflow Programming (DFP) paradigm models computer programs as graphs where each node represents an operation to be applied to the flow of events (the edges of the graph). This paradigm has been adopted by several streaming applications [13] since it represents the right abstraction for modeling event transformations. Most of these applications represent programs as a Directed Acyclic Graph (DAG), that is a directional graph with no cycles. Each edge in the graph is directed from one vertex (or node) to another such that there is no way of starting at a vertex V and follow a sequence of edges that lead back to V. The nodes can either be a source, a sink or an operational node that together form a chain. As soon as an input becomes available at the sources in the graph, the data is then sent to the next operational node that runs computations on the data and outputs the results until a sink node (with no directed edge flowing to another node) is found. This type of programming paradigm is very useful to model streaming programs, the operations in the DAG can easily be distributed in order to scale the application because each operator can be configured to function independently and maintain its own state.




```
eventSource.stream()
    .filter(event.value > 0)
    .map(event.value * 2);
```

Figure 2.6: DFP

As an example, lets assume that one wants to create a program that takes a stream of integers and filters out (i.e. discards) every negative element. After that, each element should be multiplied by two. An example of a model that represents the described program, using the DFP paradigm, is depicted in Figure 2.6. The first operator is a filter function, that is a function that has a condition to be matched against the event. If the condition is true ($value > 0$) the input element is forwarded to the next operator, in this case a Map. The Map function takes one element, transforms it and produces one result. In this case, the Map function multiplies each element by two. This complete dataflow program will keep producing even numbers as new integer values arrive.

2.4 Lambda and Kappa Architectures

The two most known architectures for stream processing are Lambda and Kappa. The purpose of both architectures is to process an immutable, unbounded sequence of events. Nathan Marz introduced the designation of Lambda Architecture [14] for a data processing system that contains both a batch processing layer and a stream processing layer which complement each other to enable the extraction of valuable information from data.

Lambda Architectures are built to process data in a series of layers - the batch layer, the speed layer and the serving layer as represented in Figure 2.7. The batch layer has two main functions: store the incoming sequence of events and compute arbitrary functions on the stored datasets. Instead of applying functions to all the data every time there is a new query to process, the batch layer offers batch views that are pre-computed queries that can be accessed by random reads. The serving layer provides means to load the batch views and makes it possible to perform the desired queries. The time it takes to update a

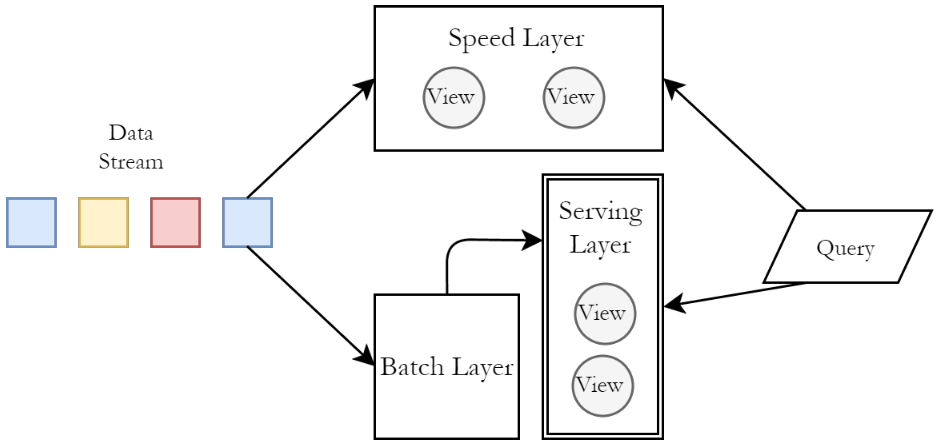


Figure 2.7: Lambda architecture

view can be of a few hours. This means that a new layer must be in place to process the arriving data in real-time. This is done by the speed layer. While the batch layer looks at all the data at once, the speed layer only looks at recent data. It updates the real-time views by means of incremental computations as it receives new data. Using the definition from [14], the lambda architecture can be summarized by the following three equations.

$$\begin{aligned}
&batchview = function(alldata)\\
&realtimeview = function(realtimeview, newdata)\\
&query = function(batchview, realtimeview)
\end{aligned} \tag{2.1}$$

The queries are made to the batch layer and the speed layer in parallel and the results are merged together.

Kappa Architectures are a simplification of Lambda architectures since they are designed to work without the batching layer as depicted in Figure 2.8. The idea was proposed by Kreps [15], who defends that stream processing can be improved to handle the full problem. In this approach, instead of recomputing repeatedly the results into batch views the reprocessing is only done when the code changes. Because this system only uses a single layer - the stream processing layer, there is no necessity to maintain two versions of the code. When there is a need to change the streaming processing job, a new job is submitted that will start to process from the beginning of the retained data. When the new job has caught up with the old one, simply stop the old version.

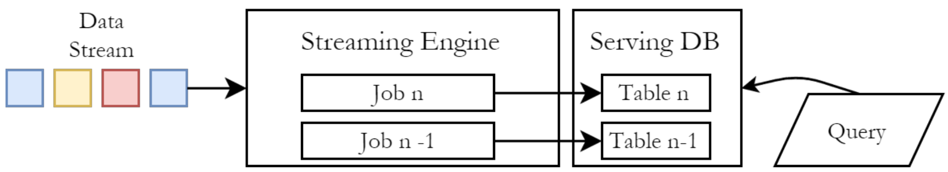


Figure 2.8: Kappa architecture

In a Lambda architecture, the collected data is fed both to a batch and to a streaming processing system in parallel. One of the arguments for forwarding the output to different engines is the inability of stream processors to produce as accurate results as batch engines and to avoid faulty results by combining the output from both layers. Although stream engines may be less mature than batch engines, these evolved into systems that can also do batch processing in a reliable manner, so the argument may no longer apply. Furthermore, because there are two processing layers (Speed and Batch) the same processing logic has to be replicated, which can be hard to maintain.

2.5 Bloom and Cuckoo Filters

Many ESP systems allow the state to be kept and managed by users. Commonly, this state is used to store auxiliary information to aid in the event analysis. For instance, one could store lists of metadata, known to indicate malicious behavior, to be matched against the incoming event fields in order to detect potential intrusions. It is important that this state is kept in memory to avoid the latency of disk reads and writes. This behavior is desired, but in most cases not achievable since the state may be too big to be kept entirely on memory. Nonetheless, there are some compact data structures that avoid storing the elements, for instance only a fingerprint of the element is stored, to occupy less space while still allowing fast lookups and insertions. An example of one of those structures is a Bloom Filter [16].

A Bloom Filter is a space-efficient data structure to represent a set of elements, which is then used to perform membership queries (i.e. to check if an element belongs to the set). By allowing a number of elements to be falsely identified as members of the set permits a much smaller hash area without increasing search time. The data structure of a Bloom filter is an array of n addressable bits that are set to 0 upon start. To get the addresses of the array, the filter uses k distinct hash functions whose values must fall within the interval $[1, n-1]$. Ideally, the hash functions should allow a uniform distribution of values across the array. To store an element e, it must be fed to each of the k hash functions which return the k array positions. All the bits corresponding to obtained addresses are set to 1, thus completing the insertion. To check if an element e' exists, one simple checks whether all $h_i(e'), i \in [1, k]$ are set to 1. If not, then we can certainly say that e' is not a member of the set. Figure 2.9 illustrates the mechanisms for inserting and checking membership of an element.

When checking if an element belongs to the set, if all the bits corresponding to the positions $(h_1 ... h_n)$ are set to 1, we cannot safely guarantee that the elements belong to the set because the bits could have been set to 1 by the insertion of other elements, therefore an element can be falsely identified as a member of the set. The false positive rate is a function of the length of the filter (m), the number of hash functions (k) and the number

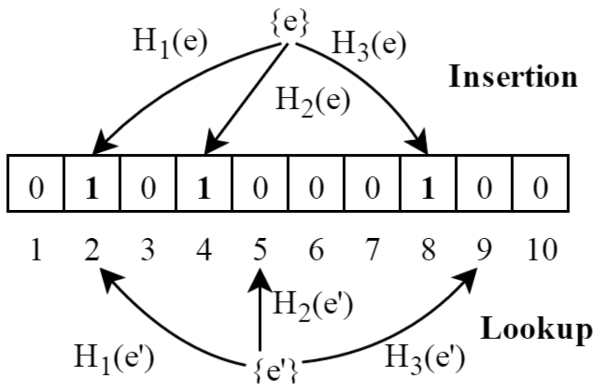


Figure 2.9: An example of a Bloom filter, representing the insertion of the element e and an inspection to verify if the element e' belongs to the set, with k=3 hash functions and n=10 addressable bits. When the element e is added to the set, the bits in the positions 2,4 and 8 and changed to 1. We can clearly see that e' is not a member because the bits in the positions 5 and 9 are 0.

of elements(n). Assuming that an hash function returns any array position with equal probability and if m is the length of the array, then the probability of a certain position in the array being set to 1 is $1/m$. Therefore, the probability that a bit is not set to one by an hash function is $1 - 1/m$. If k is the number of hash functions, then when inserting an element the probability that the bits are not set to one by any of the k hash functions is $(1 - 1/m)^k$. If we have inserted n elements, the probability that a certain bit is still 0 is $(1 - 1/m)^k n$. Hence, the probability of a false positive (all bits set to 1) is given by the following equation.

$$f \approx (1 - p)^k, p = (1 - 1/m)^k n \tag{2.2}$$

A limitation of Bloom filters is that elements cannot be removed without making false negatives (i.e. an element that belongs to the set is falsely identified as not belonging) possible. This is because we can set a location to 0 that is hashed by some other element in the set. To overcome this problem, Counting Bloom filters [17] were introduced as an extension to Bloom Filters by simply changing the array positions from bits to counters. When an item is inserted or deleted the counters are respectively increased or decreased. Since each array position is used to store a counter, Counting Bloom Filters can be 3 to 4 times larger than Bloom filters.

Cuckoo Filters [18] where proposed as an alternative to Bloom Filters which allow insertions, deletions and lockups while occupying the same or less space and have improved performance. Cuckoo Filters are a variant of Cuckoo Hash Tables [19] which consist of an array of addressable buckets where items can be stored and retrieved. The buckets for an item are determined by two hash functions $h1$ and $h2$. To insert an element e into the table, the algorithm first calculates the hashes $h1(e)$ and $h2(e)$. If either of the two buckets is empty, the item is inserted into a free bucket and the insertion completes. If neither bucket is empty, the algorithm selects one of the candidate buckets and reallocates the existing item into one of its alternate locations. This procedure may repeat until a vacant bucket is found or until a maximum number of iterations has been reached. In the latter case, the array is considered full.

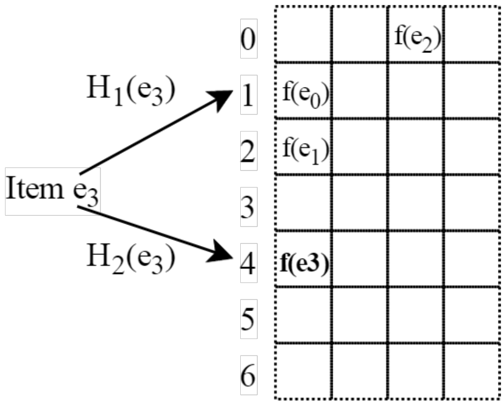


Figure 2.10: An example of a cuckoo filter with 6 distinct buckets, each capable of storing 4 fingerprints

Cuckoo Filters, instead of storing the item, only store a fingerprint (a short bit string that identifies an item). The basic units of a cuckoo filter hash table are entries, each entry stores a fingerprint. The hash table consists of an array of buckets each containing one or more entries, as illustrated in Figure 2.10. With standard cuckoo hashing, in order to find an item alternative location, it is necessary to have access to that item in order to apply the hash functions. Since Cuckoo filters only store fingerprints, there is no way to restore the original keys to find the alternative locations. To overcome this limitation, they use a technique called partial-key hashing. The two alternative locations are calculated as follows:

$$\begin{aligned} h1(e) &= hash(e) \\ h2(e) &= h1(e) \oplus hash(e'sfingerprint). \end{aligned} \tag{2.3}$$

The XOR operation assures that the alternative position can be calculated from either $h1$ or $h2$. The fingerprint is hashed to help distribute the items uniformly in the table. When constructing a filter the fingerprint size is determined by the desired false positive probability (FPP). Smaller fingerprints will incur into more collisions and thus the FPP increases. To insert an item, the process is similar to an insertion in a standard cuckoo hash table but using the Eq. 2.3. To lookup an element e, the algorithm first calculates the two candidate buckets and the $e's$ fingerprint. If the fingerprint matches an entry, the algorithm returns true, otherwise, it returns false. To delete an element, the algorithm checks if any of the entries in the candidate buckets matches the fingerprint, if so, one copy of the matched fingerprint is removed. The copy of the fingerprint does not necessarily belong to the element being removed. For example, if both elements e and e' reside on the bucket i_1 and collide on the fingerprint f, partial-key hashing ensures that they also share the other candidate bucket $i_2 = i_1 \oplus hash(f)$. When deleting e it does not matter which copy is removed because internally both e and e' are the same. Consequently, the FPP remains the same because if e is deleted, lockups for e' will return a false positive and vice-versa.

2.6 Information Security Concepts

Since the main goal of this work is to present an event processing and correlation engine that aims to detect security threats, it is relevant to understand the basic underlying security concepts. Therefore, this section introduces several information security concepts that are often referenced throughout the present work.

2.6.1 Confidentiality, Integrity and Availability

In its broad sense, security can be defined as the set of actions and resources used to protect something or someone. More specifically, information security can be understood as the protection of the confidentiality, integrity, availability and authenticity of information.

Confidentiality is the principle that guarantees the protection of information against its unauthorized disclosure. It is important to ensure this principle as it is the basis for the privacy of information inherent to an individual or organization.

Integrity can be understood as the nontampering of information in an unauthorized or undetected manner. Data integrity is assured if the system can maintain the accuracy and completeness of the data over its entire life cycle.

Availability is the property that aims to ensure accessibility and use of information whenever it is necessary. This means that the services provided by a system must function continuously.

Authenticity is the certainty that an object belongs to the sources announced. Authenticity is considered by many authors as part of the integrity principle.

2.6.2 Attack, Vulnerability and Intrusion

An Attack, Vulnerability or Intrusion (AVI) can be seen as three different fault classes as described in [20]. A fault is a defective property that leads to an error. An error is a change in the system state which is liable to lead to a failure. Finally, a failure occurs when the system delivered service deviates from its correct behavior.

Attacks are malicious attempts to violate the security properties of a system that may originate from outside the network (e.g. by a malicious attacker) or from within (e.g. by an administrator). Vulnerabilities are faults introduced by designers, intruders or operators. These can be introduced during the design phase (e.g. Improper protocol), the development phase (e.g. coding bug) or during the operation (e.g. improper permissions management). Ultimately, an intrusion is a successful attempt at exploring a vulnerability by means of an attack which may lead to a failure. The AVI model is illustrated in figure 2.11.

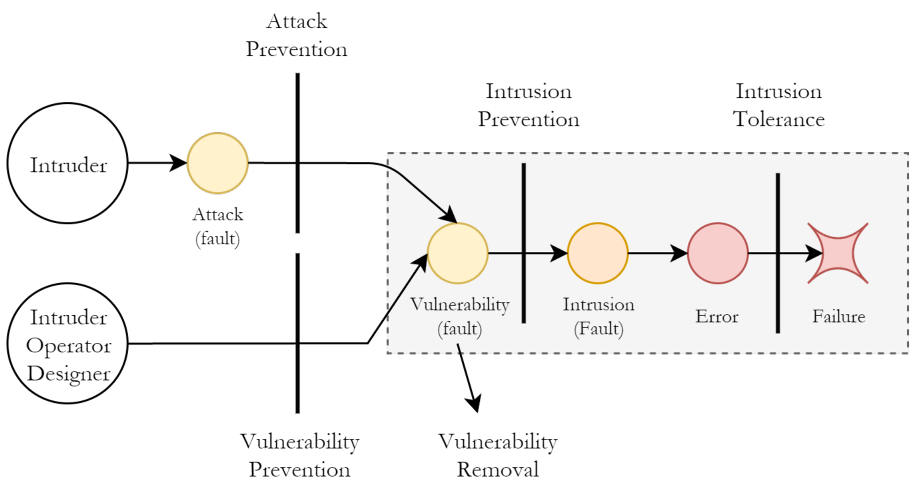


Figure 2.11: AVI model - Adapted from [20]

Depending on the fault type, different security mechanisms can be applied to prevent intrusions. We can use attack prevention mechanisms to assure that certain attacks do not take place against certain components (e.g. Firewall). Vulnerability prevention can be used to reduce a number of vulnerabilities by applying good practices in software development or measures to prevent misconfigurations. Attack Removal can be used to terminate an attack by identifying it (e.g. IDS) and taking measures to terminate it (e.g. configure a firewall). Lastly, vulnerability removal can take place to remove existing vulnerabilities by first identifying them (e.g. code analysis) and then applying the respective correction (e.g. patch).

More sophisticated systems also apply techniques for intrusion tolerance. That is, even in the presence of an error the system can provide a correct service. The error is usually masked with redundancy. That is, in a cluster of replicas, if one fails, as long as there is a majority of correct replicas, the system can continue to provide a correct service. This technique is useful in critical environments such as nuclear power plants to stop advanced persistent threats (APTs). An APT is an orchestrated attack targeting a specific identity recurring to sophisticated techniques that allow the threat to remain undetected.

2.6.3 Intrusion Detection

Intrusion detection has been the subject of much study and research in the field of information security. Instead of actively preventing intrusions, Intrusion Detection Systems (IDSs) monitor events and analyze them in order to detect possible security breaches. Modern IDSs are commonly composed by three main components: a module that gathers data, an analysis engine to process this data and identify possible intrusions and a reporting component that provides information about the analyzed data. By looking at these components, especially the analysis engine, we can easily come to the conclusion that event processing engines can be used to perform these types of analyses, but instead of logs one would have to change the processing mechanism to network packets.

There are two types of IDSs: host intrusion detection systems (HIDS), which refers to systems that reside in and monitor individual host machines and network intrusion detection systems (NIDS), which are designed to monitor a computer network. There are two distinct analysis techniques for intrusion detection: Anomaly detection and misuse detection. Anomaly detection uses models of expected behavior of users and applications, and a deviation from these models is interpreted as an abnormality. The most attractive feature of this methodology is the ability to detect attacks that were not previously identified. However false positives are usually frequent, that is, events that the IDS erroneously identifies as being intrusion attempts. In Misuse detection, one defines what is considered an abnormal behavior by means of an attack signature. A signature can be seen as a description of what an attack may be and an intrusion is detected when there is a match between the audited data and a signature. This mechanism is usually very accurate and hence the number of false positives is commonly low, but it is incapable of detecting attacks that are not present in its knowledge base (i.e. within the set of attack signatures).

An attack signature is a fundamental piece in an IDS. The signatures contain the patterns that need to be identified when monitoring network traffic. For instance, an attack signature may be a subject of an email plus some other metadata that identifies an email potentially containing a virus. Attack signatures are a subset of Indicators of Compromise (IOC) which represent records that with a high degree of confidence may indicate an intrusion. An IOC can be a list of email addresses, domain names, IP Addresses, URLs and other artifacts. From a security standpoint, it is very important to have a database of trusted and up to date IOCs so that potential attacks can be quickly identified by comparing the event data with the lists of IOCs.

Chapter 3

Event Processing Frameworks

The aim of this chapter is to identify, describe and compare some of the existing event processing frameworks so that we can choose the one that best suits our needs. The description of each studied framework is presented in Section 3.1. Section 3.2 describes the criteria used to compare the solutions. Section 3.3 presents a comparative analysis of the frameworks using the defined criterion, describes our testing environment used to further evaluate some of the frameworks and presents the conclusions derived from the frameworks evaluation.

3.1 Frameworks Description

In the last few years, there has been a lot of interest around ESP and CEP systems with several proposed solutions. Systems such as Esper [21], Drools [22] and WSO2 [23] process data streams by running continuous relational queries that rely on languages derived from SQL. These are used to express the conditions where an event or a group of events are of interest while hiding the low-level mechanics from end users. Solutions like Spark [24], Storm[25] and Flink [26] provide an high-level API which allows users to build their own processing logic. Although systems that rely on EPLs are generally easier to use and maintain, they often lack the desired flexibility to develop more tailored solutions. On the contrary, systems that provide high-level APIs allow users to build custom modules but at the same time are more complex to develop on.

Our objective is to bring together the best of both worlds, providing a flexible system that is capable of adapting to different scenarios and, at the same time, is capable of providing an easy to use language and/or user interface.

Spark

Apache Spark is a general purpose event processing engine that is capable of both batch and stream processing. Spark, aims to make data analysis faster and simpler by providing several libraries, including machine learning, computation on graphs and an SQL like language to query datasets. It also provides programming interfaces in various languages such as Java, Scala and Python. Spark requires a cluster manager and a distributed file system to manage its replicas and maintain a coherent state. Usually, Hadoop YARN [27] is used as the cluster manager and Hadoop Distributed File System (HDFS) to store and share state across several nodes.

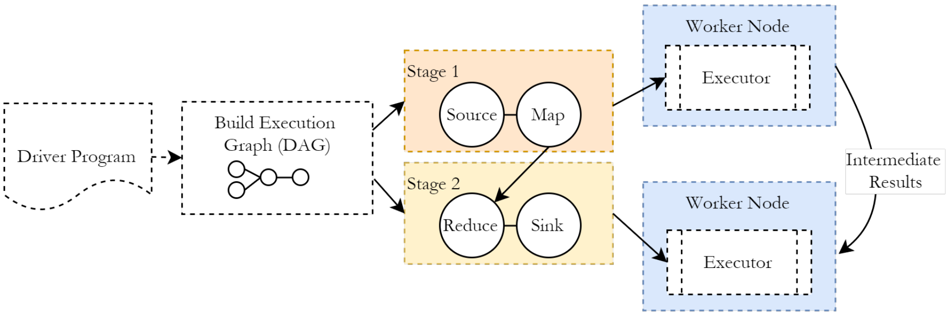


Figure 3.1: Spark's distributed execution

To create applications in Spark a user needs to create a driver program, where he specifies how the data is collected and transformed using several operators. The various operations contained in these programs can be executed in parallel. For parallel execution, Spark uses a concept called Resilient Distributed Datasets (RDDs). RDDs are fault-tolerant, parallel data structures that can be manipulated using a vast set of operators. A user can collect data from an external source and specify how the data is going to be transformed or aggregated using RDD operations, some are functionally similar to the dataflow programming functions discussed in Chapter 2 (e.g. Map, Filter). The state of an RDD is stored in memory as an object that can be shared across several tasks. When the code (driver program) is submitted for execution, the interpreter builds a DAG that is essentially a chain of RDDs. Then, according to the dependencies between operations, the execution graph is divided into stages that are passed on to worker nodes that execute the tasks. This workflow is illustrated in Figure 3.1.

To run on a cluster, Spark's main program (or driver program) connects to a cluster manager which allocates resources across applications. Once connected it acquires the executors that are present on each worker node to run computations on the data.

Flink

Like Spark, Flink is an open-source system for streaming and batch processing. Flink programs embrace stream processing, making no distinction between processing events in a one-by-one fashion, using window aggregations or historical data. These types of computations are made using the same underlying mechanisms, with the only difference of starting and ending at different points. Flink provides programming interfaces in Scala and Java. Unlike Spark, it can be deployed using its own cluster management system.

The building blocks of a Flink program are data sources, streams, transformation operators and sinks. A source is the place where Flink will get its data from. A stream represents the flow of events. Transformations are operations applied to the data streams, whether corresponding to unions with other streams, filtering unnecessary events, aggregations and aggregation functions. Finally, sinks represent the output of the data.

The complete flow of data from the source to a sink is called a Streaming Dataflow and an example is represented in Figure 3.2. The dataflows can be thought as DAGs where at each stage of the graph new transformations are applied in order to change the state.

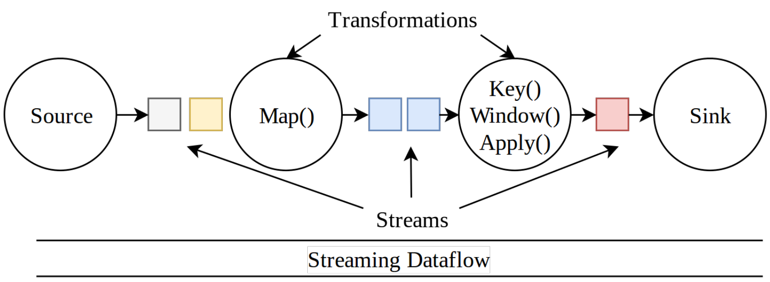


Figure 3.2: Flink's Streaming Dataflow

Flink operations are highly scalable. One of its features includes a support for changing the parallelism level of each operation. This means that each node in the DAG can be distributed across several tasks that would share some state, which allows them to perform the parallel execution. For instance, if we set up a level of parallelism of two for the first three operator nodes in the DAG and a level of parallelism of one for the sink node, the operations would be distributed as Figure 3.3 shows.

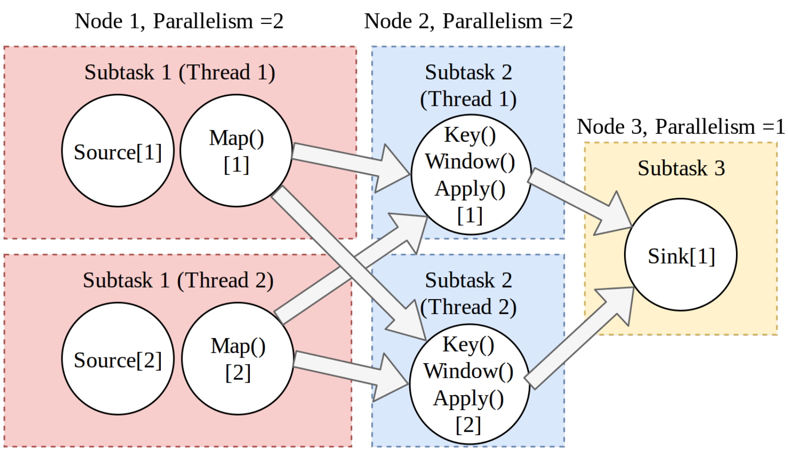


Figure 3.3: Flink's Parallel Dataflows

Flink follows the same Master/Worker approach as Spark and it is composed of three different types of nodes: Job Managers (master nodes), Task managers (worker nodes) and Clients. Job Managers coordinate the distributed execution, task managers execute operations and the clients are used to submit user programs for execution.

It is clear that Flink's workflow is very similar to Spark's. Both gather data from a source, apply transformations to that data, in RDD form or data stream and send the results to a sink. The main difference between Spark and Flink is at their core processing model. While Spark was initially built as a batch processing engine, Flink was built as a stream processing engine. This means that Spark only emulates streaming while with Flink we can build real-streaming applications. The way that Spark emulates streaming is by issuing micro-batches. A micro-batch is a very small time window that collection of events to be processed. Window constructs and operations can still be made in spark by iterating over a series of micro-batches, however, this is more expensive in terms of processing resources. In Flink as soon as an event arrives a computation can be carried out, while in Spark there is a small latency while waiting for a micro-batch to complete.

Drools

Drools is a Business Rule Management System (BRMS), which is a software that can govern and automate the decision logic within an organization. More importantly for us, it has a CEP library named Drools Fusion. This open source library allows users to process events by modeling business rules or queries that are matched against the incoming events. In order to model these rules, Drools provides its own EPL. Drools Fusion can also be integrated with other products from the Drools ecosystem, which allows the construction of a close to complete solution.

Drools has two types of memory, the production memory where the rules are stored and the working memory, where the facts (the data to be analyzed) are stored. The production memory is fixed and cannot be changed without stopping the currently running application. On the other hand, the working memory can be dynamically modified. The Drools engine works with these two memory sets to match the rule patterns against the facts. When a user wants to create a new set of rules to analyze some data, it programs a java class to collect both the data and to compile the rules. The rules have to be created using Drools own EPL and are fed to the java program that compiles them and matches them against the data.

Unfortunately, Drools follows a centralized architecture, which means that the computations cannot be distributed. Moreover, by reading the documentation there is not much information about its architecture and working examples.

Esper

Esper is an open-source framework that allows for CEP using a programming interface in Java and, like Drools, it also provides an EPL to express the conditions where an event is of interest. Esper's interface and EPL allow users to build complex rules using very simple semantics.

Esper has its own EPL, which allows the expression of filtering conditions, aggregations, joins, sliding windows and other important constructs. It also includes pattern semantics to allow the expression of temporal causality among events. Contrary to Spark and Flink it does not have a distributed architecture but it can be integrated, for instance with Storm, to build a distributed engine [28].

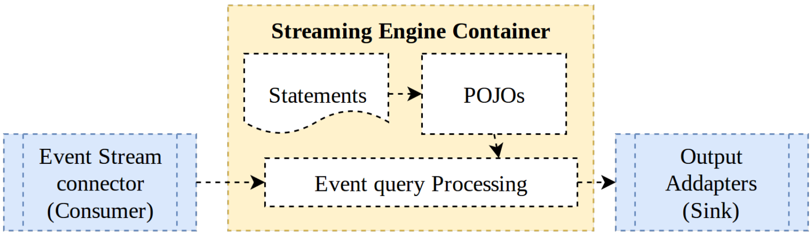


Figure 3.4: Esper - Architecture

Esper's architecture is illustrated in Figure 3.4. It has the possibility to access streaming data from event stream connectors. Esper's engine has a series of statements which hold the logic of the program written with their EPL. All the data is expressed as a POJO (Plain Old Java Object). For instance, all the firewall related events can translated to a firewall object that maintains the log fields as variables. As soon as events arrive they adopt a POJO model to be matched against the created statements. In our case, the statements will represent the set of security rules.

Storm

Storm is an open-source framework that allows for distributed computations on unbounded data sets. Storm makes distributed processing simply by providing an easy to use interface. Moreover, it is reliable, scalable and easily deployable. Despite its powerful distributed and processing engine, Storm is usually used alongside other processing applications due to the lack of powerful constructs and functions for event correlation.

The basic storm data processing architecture are streams of tuples that flow through topologies. Like Flink's data flows or Spark's DAGs, a topology is a graph where vertexes represent computations and the edges the data flow between the different nodes. Storm makes a distinction between vertexes that pull data from sources, named spouts, and vertexes that process the incoming tuples, the bolts. Figure 3.5 shows a simple topology that counts the words in a stream of lines. This topology has one spout and two bolts. The spout pulls lines from a data source. One of the bolts receives those lines and splits them into words. The final bolt counts the number of words.

Clients, create their topologies using Storm's Java API and then submit them to a master node, which is called the Nimbus. The Nimbus is responsible for managing the worker nodes that in turn perform the operations. Each worker process is mapped to a single topology and has one or more tasks which perform the spout or bolt operations.

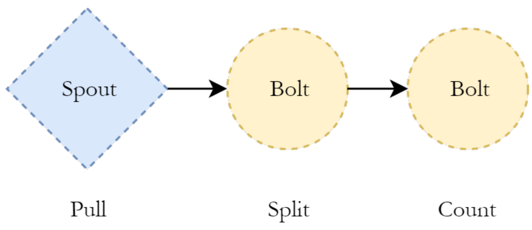


Figure 3.5: Storm topology - Word count

WSO2

WSO2 provides a wide range of products for message delivery, software integration and analytics. The one that we are most interested in is the WSO2 CEP, which is based on Siddhi [29]. Siddhi is a lightweight CEP engine that provides an easy to use interface in Java as well as an EPL similarly to Drools. WSO2 improved on Siddhi adding several new features both to the EPL as well as to the interface. This solution is highly scalable and also provides a web interface that allows users to easily create their own processing modules.

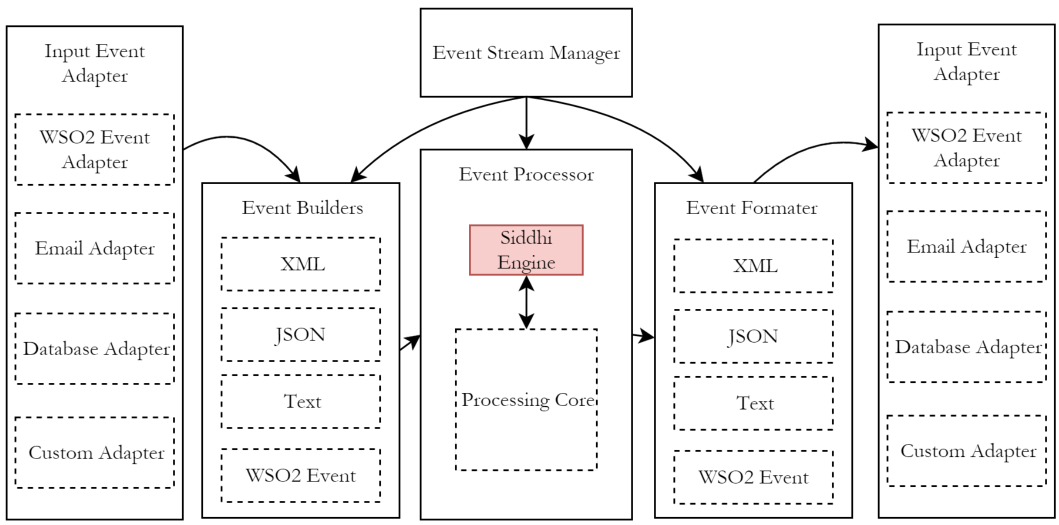


Figure 3.6: WSO2 Architecture

Using WSO2 a user can submit execution plans to the CEP engine that does the actual processing. The execution plans are similar to the execution graphs for the other engines. WSO2 architecture is composed of 6 main components. The Input Event Adapter that receives events in a raw format (as they're coming from the source). The WSO2 has several custom adapters to connect to various sources (e.g. Email, Databases, etc). Once these events are collected, Event Builders convert those events to XML, JSON or other optional formats to be processed internally. The event processor is the core of the solution, it receives and executes the plans. The Event Formater converts the event streams to an output format to be published by the Output Event Adapter. Finally, the Event Stream

Manager coordinates all the components and provides them with information regarding the streams. Users can also submit new execution plans through this component.

3.2 Definition of the Comparison Criteria

In order to evaluate the described frameworks, we first created a list of functional and non-functional requirements that our system must fulfill. It is important to mention that these requirements highlight the characteristics that our system ought to have after the design and implementation and not the requirements that the framework should have. Nevertheless, these are important to define the key criteria to be used when comparing the frameworks. It is important to emphasize that the evaluation is not meant to be a detailed guide that compares every aspect and addressed problems of each framework but instead focuses on some key points that are important to our final solution.

3.2.1 Functional Requirements

The functional requirements describe what features should be provided by each of the components of the system. Our system functional requirements are described as follows.

Event Collection: The system must be able to collect the logs from different systems using push (the component sends events to the event correlation engine) or pull techniques (the correlation engine retrieves the logs from the components).

Dynamic Rules: The system must allow the creation, deletion and edition of security rules at runtime. The rules are constantly changing both due to the discovery of new vulnerabilities and due to the necessity of monitoring new components that are added to the infrastructure.

Filtering of Events: The security rules must allow the expression of filtering conditions. The components being monitored continuously produce log data that is collected and then parsed. The parsed logs represent our events that should be filtered by means of a boolean expression that contains the set of conditions which indicates if the event is of interest and thus should be further processed.

Aggregation Functions: The security rules must allow the expression of aggregation functions such as an average function to a collection of elements that belong to a certain time window.

Pattern Operations: In order to detect more sophisticated and complex attacks, the system must allow users to apply pattern functions to a collection of elements.

IOC Lists: Our system must be able to maintain a set of IOC lists so that an incoming event may be matched against them to detect malicious behaviors.

Establish Metrics: It is important that the system outputs some metrics that provide an insight of what is currently happening. Some examples of these metrics can be the number of incoming events per second, the top source addresses that access a certain server, the top accessed URLs and others.

3.2.2 Non-Functional Requirements

The non-functional requirements describe the operational characteristics that the system should have, rather than specific behaviors.

Availability: Since an attack can quickly spread and have disastrous consequences the system must present a high availability with minimum interruptions so that if an attack is performed it can be quickly detected. Although we do not yet have the exact availability metrics that we want to obtain, we aim to develop a system that can be continuously running in advert circumstances, such as when a user incorrectly uploads a rule or there is a fault in one component. Furthermore, since we are dealing with data that may contain insights of possible attacks and their perpetrators its availability is a must.

Non-Intrusiveness: The data collection must be transparent to the components being monitored. That is, without any significant changes and side effects to the applications being monitored. Since the components to be monitored are critical we do not want to cause unnecessary downtime.

Performance: The number of events produced by the infrastructure components is massive. As such, the system must provide a high throughput. Moreover, the amount of time between the beginning of an attack and its detection is crucial. If it is too long, the attack may be further propagated, so low latency is a must.

Security: The data that enters or leaves the system may contain confidential information, so it is important to guarantee that the communication between the sources, system and the sinks is secure. Furthermore, only allowed users should be able to access the system, which requires some kind of user authentication.

Versatility: A user should be able to create a rich set of security rules that are able to express all the necessary conditions for an attack or a set of attacks to occur.

Extensibility: The system must take into account future growth. We do not know what the future might hold and at any time we must be able to add new functionality.

3.2.3 Criterion

Each criterion describes the ideal scenario that would satisfy the requirements for our end-system. Naturally, the framework will not fully satisfy each criterion, therefore we also describe the key aspects of our evaluation.

Table 3.1: Criterion

Criteria	Description	Key Evaluation Aspects
Features	The features are the functional characteristics that the system is able to offer.	1. Is the framework capable of collecting data from different sources, possibly with different data formats (e.g. JSON and XML). 2. Does the system provide enough features to allow us to develop a module to analyze event data. 3. Can the framework be easily integrated with other tools.
Ease of Development	Ease of Development is the capability that the software has to enable the developer to easily understand and use it.	1. Is the documentation available and easy to understand. 2. How easy it is to develop applications using the provided the APIs or interfaces.
Performance	The performance is the amount of work that can be performed by the system.	1. Can the engine provide a high throughput. 2. Can the engine provide low latencies.
Scalability	Scalability is the ability that the system has to handle a larger amount of work.	1. Is the engine able to scale horizontally (i.e. can the application distribute its state or computations apply across different machines). 2. Is the engine capable of increasing its performance once given more computational power.

Fault Tolerance	The fault tolerance is the ability that the system has to deliver a correct service even in the presence of faults. The ability for a distributed system to remain available and consistent is directly linked with its fault tolerance guarantees.	1. Does the system have mechanisms to handle faults in order to continuously deliver a correct service.
Licensing	A software license is an official permission to alter and/or use it under certain conditions.	1. Can we freely use all of the provided features. 2. Can we freely alter the system code and use it to fit our needs.
Community	Community is everyone involved, directly or not, in development and usage of the system and that may provide help and insights about it.	1. Does the framework maintain an active community in forums such as stack overflow.

3.3 Frameworks Assessment

In this section, we provide the results of our evaluation using the previously defined criteria. This is not a thorough evaluation that scientifically compares every aspect of the frameworks, instead, it focuses on some key aspects that are relevant to the development of our system and may be of interest to other researchers trying to develop similar platforms. The conclusions described in this chapter regarding the discussed criteria were based on the documentation of each framework together with some other existing works [30, 31, 32].

In terms of features, Drools, Esper and WSO2 stand out as fully fledged solutions. Not only do they have a full featured streaming engine that offers CEP capabilities, but also many other complementary products which provide a rich set features like a user interface and additional security mechanisms. Flink and Spark offer similar libraries which include batch and stream processing libraries, machine learning, computations on graphs and others. Storm, offers a more specialized engine that only supports distributed stream processing with few features for complex event analysis. All of the frameworks provide some way to connect to third party software and collect event data in several formats. However, WSO2 stands out because it has the most connectors and Esper falls short because of the lack of support to integrate with some modern platforms such as Kafka [33].

In terms of ease of development, Drools, Esper and WSO2 are the easiest to develop on. They shifted towards more enterprise use, as such their focus is more on stability, ease of development and rules creation. However, they only support Java and lack flexibility for developers to easily change and adapt the mechanisms to their needs. All of the other frameworks offer similar APIs in Java, Scala or Python that are flexible and allow for a relatively easy development. As for the documentation, with the exception of Drools all are easy to navigate through and understand.

As for performance, we did not found much information in the documentation. All frameworks claimed a good throughput and low latencies, but according to some benchmarks [30] Flink stands out because it offers both the highest throughput and the lowest latencies.

Regarding scalability, all the frameworks scale horizontally, with the exception of Esper and Drools which only scale up (i.e. they are not able to perform distributed computations), which is a major disadvantage.

The fault tolerance guarantees differ for each engine and are hard to validate since we would have to test them in several scenarios and there are not existing works which do so. Nevertheless, all of them claim to offer at least node recovery. By reading the documentation we could not find out much information about Drools and WSO2. However, we do know that Spark, Flink and Storm offer exactly-once semantics and checkpoints to allow the state to be correctly recovered after a failure. Esper offers the solution EsperHA [34], which has mechanisms to provide high availability.

In terms of community, all maintain an active community in stack overflow, mailing lists or forums with the exception of Esper. Spark and Flink stand out both because they have a great community and present signs of future growth due to the interest that some widely known companies have for them, namely Facebook, Netflix and Intel.

All of the studied frameworks are open-source, although some of Esper's and WSO2 require additional licenses for some complementary products.

After our initial assessment, we believe that Flink is the framework most adequate for our needs. Despite being very recent, it has already caught a lot of attention, having as contributors Cisco, Intel, Netflix and other renown companies. It supports development in Java, Scala and Python and as low-level mechanisms which allow users to program specific behaviors to extract the most out of the event data. Moreover, it has a dedicated CEP library for detecting patterns and its steam-processing engine is acclaimed to be one of the fastest. Spark is also a very powerful tool worth considering. It has a great community and many deployed solutions in widely known companies such as Facebook, Yahoo!, Netflix and many others. Although WSO2 offers a very good and complete solution, it lacks the flexibility for us to develop our own tailored system. As for Esper and Drools, they do not have the ability to scale out on their own, which means that we would have to integrate them with another distributed framework, and hence extra coding and debugging work would have to be carried out, which is not desired. Moreover, they do not have such active communities as the other engines.

Although we believe Flink to be an adequate framework to develop our system, we did not want to discard the others right off the bat. Hence, we created a demonstration application using Spark and Esper besides Flink. Spark was chosen due to its already mentioned capabilities. Esper, despite the lack of horizontal scalability, was one of the first CEP tools created, maintaining a lot of important features. Moreover, we wanted to study its EPL which seemed very intuitive for end users. In order to perform this second step assessment, we created a proof-of-concept (POC) architecture which allowed us, not only to pave the groundwork for the final solution but also to ease the creation of the demonstration application for a more detailed comparative study. Figure 3.7 illustrates the POC architecture which follows the design ideas of Kappa architectures, due to their simplicity.

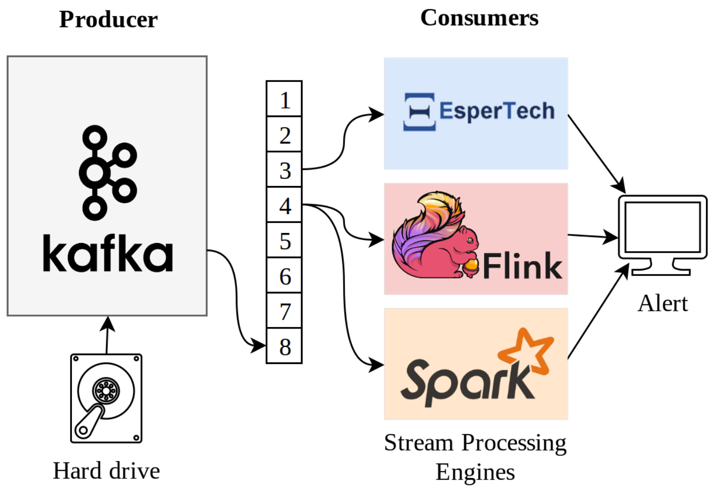


Figure 3.7: POC architecture

Instead of reading log data directly from a file, we decided to use a message broker named Kafka [33]. This allowed for two things: to see how easily could the framework be integrated with other tools and to test a messaging system which provides some useful guarantees that could be used in our final solution. All of the consumers read the same sample set of events from Kafka, apply the respective computations and write the results to the console for we to compare them. The next sub-sections describe in more detail the architecture components and the demonstration rule that was used in all three engines.

Event Source

In order to provide streams of data to applications, Kafka creates an abstraction called a topic. A topic can be seen as a feed to where data can be published to be later consumed by the subscribers. In our case, the data that is being published to the topic is a Comma Separated Values (CSV) file containing sample log lines and the subscribers represent the streaming applications. To maintain a sequence of ordered events, each record is assigned a sequential id number called the offset. The consumers need only to store this value in order to know which should be the next record to be consumed as illustrated in Figure3.8.

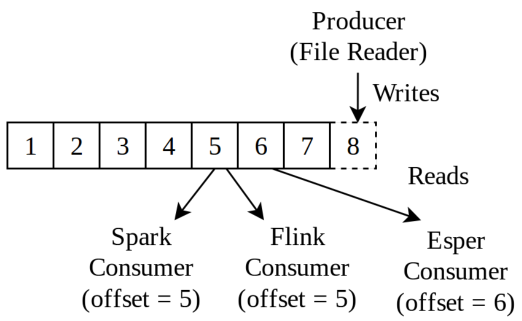


Figure 3.8: Kafka - Consumer Offset

Kafka presents several guarantees that make it particularly useful. First, different consumers reading from the same non-replicated topic will always process messages in the same order. Second, Kafka can be setup as a cluster where each partition represents an ordered, immutable sequence of events where data is continually appended to. For the coordination of the nodes and leader election Kafka uses Zookeeper [35]. Ultimately, data sent to Kafka is written to disk and replicated for fault-tolerance.

Kafka design and architecture not only makes it a very reliable messaging system but also very easy to use. Producers only need to worry about writing to the correct topics and consumers only need to maintain the offset value.

Demonstration Rule

In order to compare the engines in terms of ease of development, we implemented the same rule in each engine. The example rule sends an alert to the console if a certain user had its access denied to a server more than 10 times in a 5 second period. The pseudo-code is illustrated in Listing 3.1. This rule allowed us to test some of the most important mechanics, which are window aggregations because we need to keep collecting 10 seconds worth of events, aggregation functions to count every pair (user, destinationaddress) and a filter condition to only filter in denied actions.

```
RULE "Possible Brute Force"
  SELECT user, dstAddr, action
  WHERE action == "denied"
  AGGREGATE BY user, dstAddr
  HAVING COUNT(*) > 10
  WITHIN 5 seconds
```

Listing 3.1: Demonstration Rule

Assessment Conclusions

The environment configuration was intuitive and similar in all cases. As for the connection with Kafka, both Spark and Flink provide libraries that allow for an easy integration with Kafka as well as other frameworks like Hadoop and Elasticsearch, while Esper falls short because the consumer had to be manually programmed. As for the program logic, this is where Esper shines the most due to its simple and expressive language, which allowed the creation of the rule in half the number of code lines in comparison with Spark and Flink's native API. The program samples from each framework can be found in appendix A.

Although Esper has a lot of potential it also carries many flaws. To begin with, it does not scale horizontally. In order to do that we have to integrate it with another engine. Moreover, despite being released in 2006 Esper has a very small community. As an example, it only has 316 questions tagged on stack overflow. In contrast, Flink was released in 2015 and has already 639 questions tagged, as of 28 November 2016. As for Spark, the fact that it emulates stream-processing could present latency problems when dealing with high volumes of events. Due to the issues mentioned and after a pondered consideration, we decided to move forward with Flink as the base engine for our system.

Chapter 4

Apache Flink in Depth

In order to design our system so that it fulfills the requirements identified in Section 3.2 an in-depth analysis of Flink's software and execution environment is essential. This study not only provides a better understanding Flink's architecture and API but also identifies the lack of some features that need to be overcome.

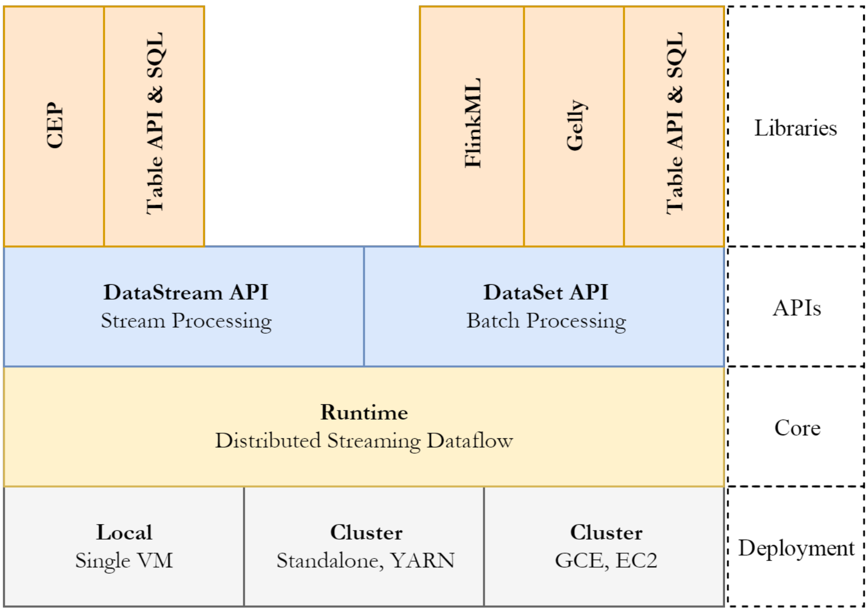


Figure 4.1: Flink software stack - adapted from [26]

Flink's software stack is composed of several layers, as illustrated in 4.1. Each layer raises the level of abstraction of the previous, which grants developers the power to program complex applications without having to worry about all the underlying mechanisms.

We will not delve into a detailed explanation of the *deployment Layer*, other than to say that applications can be deployed in three different ways: locally (mainly used for testing purposes), in a cluster or in the cloud.

The *runtime layer* constitutes the basis for all the other software components and libraries. A detailed explanation of its underlying mechanisms is given in Section 4.1.

Since we developed our application mainly using the *DataStream API*, we discuss some of its features in Section 4.2. The remaining libraries are built on top of the *DataSet* or *DataStream APIs* and provide a higher level of abstraction that allows developers to easily develop applications that deal with complex problems, such as machine learning.

4.1 Runtime

At its core, Flink provides real Stream Processing, meaning that the data is processed an event at a time rather than in batches. All the programs developed in Flink use the same runtime environment, which enables many of the performance features. Job Graphs, an important abstraction of the runtime time environment, as well as its distributed architecture, are explained in detail in this section. Additionally, we also provide some insights on how Flink handles time.

4.1.1 The Job Graph

All the programs from higher level APIs are translated into Job Graphs, which represent the complete data flow of the program. The Job Graph consists of several operational nodes (or tasks) that together form a DAG, as described in Chapter 2.3. Each node in the graph produces intermediate results that are forwarded down the pipeline until they reach a sink. Since each node is designed to function independently and maintain its own state, it represents the ideal abstraction because it can be deployed as its own entity in a distributed system. A node needs only to worry about correctly receiving the messages from the source nodes and sending the (possibly transformed) events to the correct destination nodes. A typical topology consists of source nodes that gather and parse raw log data to an internal event format, a set of transformation nodes that apply computations to those incoming events and sinks that forward the events to the desired destinations.

To allow the applications to scale using this abstraction, Flink translates Job Graphs to a parallelizable version called an Execution Graph. An execution graph can have several replicas of the same operational node that can be distributed between different machines. For instance, a node could have a level of parallelism equal to five, which means that the Execution Graph will have five execution nodes (or vertexes) just for that one operator. This implies that data can be split among the five execution nodes and be computed in parallel (as long as there are no dependencies between the data sets). Flink jobs and operations are very flexible, for instance, the level of parallelism can be set arbitrarily for each node by the developer of the job. Additionally, the partitioning of data to each node can also vary depending on the developer specification. For instance, the data of a source node can be subdivided to every destination node or, alternatively, it can send the whole set (i.e. an exact copy) to each. Depending on the operations it may be necessary apply use the former of the latter technique. For instance, one may want to replicate the data across several nodes for redundancy where the results are continuously compared or, for faster processing times, split the data and combine the results whenever possible. Windows are a special case because data is split according to the specified keys so that the receiving nodes contain all the events that belong to the same window. There are also several other applications that have different requirements and which may have different types of partitioning for each node in the Job Graph.

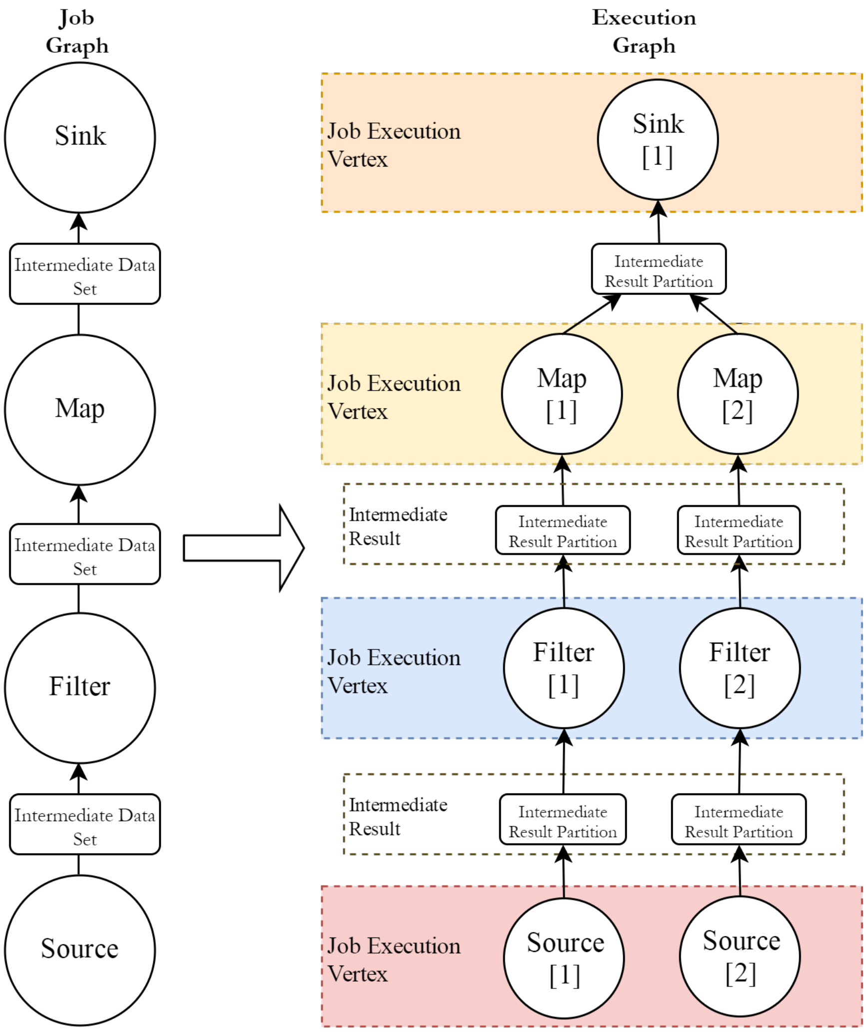


Figure 4.2: Job Graph translation to an Execution Graph

In terms of performance, the partitioning of data has some advantages and disadvantages depending on the situation. When an operational node or task has heavy computations but a low or medium number of input events it is preferable to split the data across two or more nodes. On the other hand, if the computations are not very demanding, processing all the events locally is advantageous since the data does not have to be sent over the network to other nodes, creating additional traffic. Besides the execution nodes, the Execution Graph contains intermediate results and intermediate result partitions. The former tracks the state of the Intermediate Data (the data produced by an operator), the latter the state of each of its partitions. This design, allows developers to program streaming applications by only specifying the desired behavior of each node and its level of parallelism while the engine takes care of the rest. An example of a job graph and its translation into an execution graph is depicted in Figure 4.2. In this example, the job was configured so that each operator/task has the level of parallelism set to 2 except the sink.

One of the problems with the Job Graphs is that they cannot be changed dynamically (i.e. at runtime). Every time a user needs to add new operators to the running job, it needs to reprogram the job, compile it, re-submit and stop the old one. Ideally, the job should be kept running while users modified the behavior of operational nodes. Unfortunately, this feature is not provided.

4.1.2 Distributed Communication

Flink is a distributed system designed to be highly scalable so that applications can easily grow to handle a higher number of events. The control flow messages between all Flink processes is done via Akka [36]. Akka is a framework to develop robust distributed applications, that is, concurrent applications that are both fault-tolerant and scalable. The Akka framework is an implementation of the Actor Model [37]. The base primitive in this model are *Actors* that communicate with each other by exchanging messages. Each actor has its own isolated state containing a mailbox in which the received messages are stored. An actor has a single processing thread that polls the messages from the mailbox and processes them successively. In response to a message, an actor can: make local decisions, spawn new actors or send new messages. The exchange of messages is done asynchronously. An Actor System is a container for all the actors, which provides services like scheduling, configuration and logging. An Actor System is capable of communicating with other remote systems via network and locally through shared memory. Data exchange between tasks (data streams) is made over the network using object serialization.

Flink distributed architecture consists of three components that communicate via the Actor Model: The JobManagers (master processes), the TaskManagers (worker processes) and the JobClients that submit programs for execution. Figure 4.3 depicts the interactions between the components, whose roles are described as follows.

JobManagers: Master processes that coordinate the distributed execution by scheduling tasks, checkpoints, coordinating recoveries on failures, amongst other tasks. Master nodes upon receiving a Job Graph create an Execution Graph that contains the tasks that have to be deployed to the worker nodes for execution.

TaskManagers: Worker processes that hold the state of the application, execute the transformations on data streams and exchange data. The Task managers are the ones responsible for receiving the execution nodes (the parallel instances of operational nodes of the job graph) as tasks and execute computations as data flows in.

JobClients: The client is used to submit jobs to the master but it is not part of the execution environment. It can also retrieve status updates such as the number of jobs running, the number of job managers and task slots, etc.

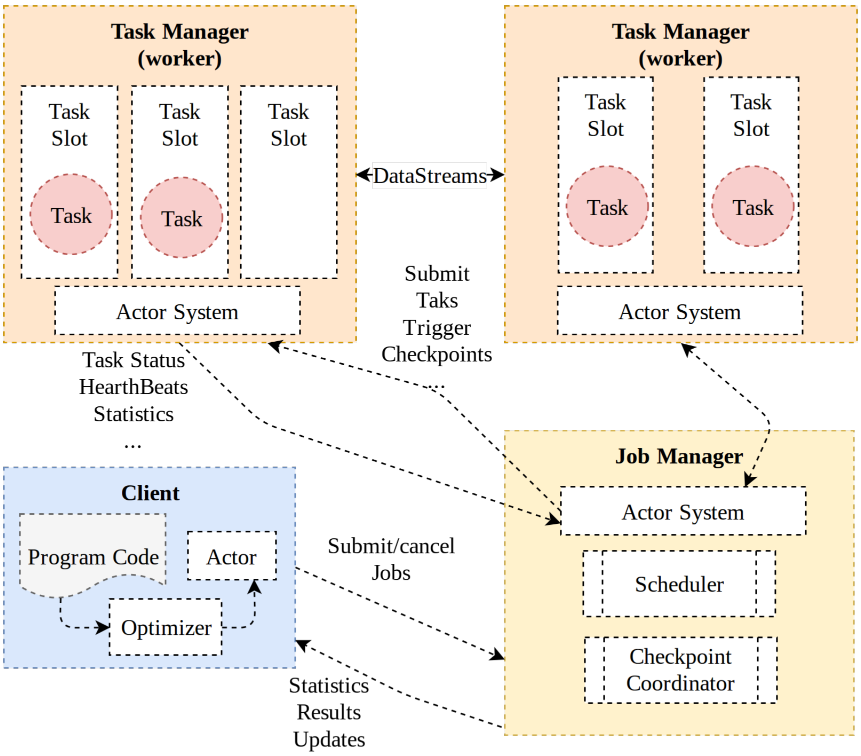


Figure 4.3: Flink's high level architecture

Flink has to be started with at least one Job Manager and one Task Manager. A high availability setup will have multiple master processes, one which will be the leader. The more worker nodes and slots per worker the more parallel operations can be successfully executed. Worker nodes must be registered at the Job Manager, which will allocates the resources. Each worker is a single Java Virtual Machine (JVM) process that contains one or more Task Slots. These slots represent a fixed set of resources that will execute tasks. A worker with three slots will dedicate 1/3 of its memory to each slot. Adjusting the number of worker nodes and task slots per each node has different advantages and disadvantages. Having few task slots and several worker nodes means that tasks become more subdivided between distinct Task Managers and thus more isolated. This is useful to ease the burden on each worker, but it also implies that more messages have to be exchanged (possibly over the network). Having fewer worker nodes with more tasks will allow processes to share resources because tasks on the same node will share the same JVM, but the workers may become overburden. In order to optimize the performance of the job, the level of parallelism has to be correctly adjusted to the set of task slots. A job with a large number of nodes and a high number of parallelism will naturally require more resources and thus more task slots available. As a general rule of thumb the number of task slots of each task manager should be equal to the number of CPU cores available.

The Job Clients are responsible for submitting the job to be then executed by the worker processes. Upon receiving a job, the Job Manager will send the individual tasks to the worker processes. Upon receiving a task, the worker spawns a new execution thread. Some statistics regarding the execution of the tasks are then sent back to the master process that in turn sends them to the client process as statistical data.

Currently, there is one evident problem with Flink's distributed architecture. One cannot choose which particular execution nodes get assigned to which machine. Lets say we know beforehand that one of the nodes in our Job Graph will perform heavier computations than the rest. Due to this reason, one would be to set a high level of parallelism to that node. The problem arises when there are distinct task managers, ones running on high-end systems and others on low-end machines. Since one cannot choose to whom the Job Manager will assign the execution nodes, the heavier tasks could end up on the low-end machines which can be problematic in terms of performance.

4.1.3 Time and Windows

In Flink, windows can be time driven (Temporal windows) or data-driven (Count windows). Count windows are easily created as the operator simply needs to divide windows by counting the number of elements. Temporal windows are more complex and error-prone. Flink supports different notions of time: Processing Time, Event Time and Ingestion Time.

Processing Time: All time driven operations (e.g. windowed constructs) will use the clock of the machine that is running the task. This is the simplest notion of time, but it does not provide accurate results since the processing time can vary from machine to machine and the clocks are not synchronized.

Event Time: The time driven operations will use the time embedded in events. Event time gives the most accurate results as the operations do not depend on clock synchronization. An hourly time window will collect all the events whose timestamp falls into that hour, regardless of when the records arrive and in what order.

Ingestion Time: All the time driven operators refer to the time the event enters in Flink. Compared to Processing Time it is more expensive but gives more predictable results and compared to Event Time programs cannot handle any out-of-order events.

Ideally, event time and processing time would be equal, which would mean that events are processed as soon as they are generated. However, this time interval can vastly vary and the skew can be of a few seconds to several minutes. This time variation can be due to network latency, the time it takes to process an element before forwarding it to the next operator, software errors and other factors.

In order to use event time, there needs to be a mechanism that keeps track of the progress of time. For this, Flink uses Watermarks, which flow as part of the Data Stream and hold a timestamp. When a $Watermark(t)$ is emitted, the time has reached t in the stream. This means that all events with $t' < t$ should have arrived. This mechanism is illustrated in Figure 4.4. It is possible that certain elements violate this condition, which is very common since it is not possible to predict the time at which an event whose timestamp is between the last emitted watermark and the next will arrive. In order to avoid discarding late events, one can define a lateness interval (e.g. 2 seconds) so that events are stored in memory for a period of time before getting discarded. This allows late events to be matched against events generated in the same time interval.

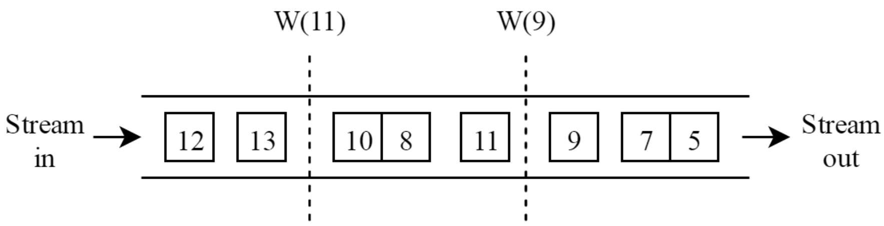


Figure 4.4: Watermarks progress

Since tasks will be performed on different nodes, each task much share the watermark values in order to advance the time correctly on all operators. Watermarks can be generated at source functions or right after them. Whenever an operator advances its event time, it generates a new watermark and sends it downstream to the respective successor operators. Operators that consume multiple input streams must track event time from distinct input streams. The current operator event time is the minimum of the input streams event time so that the time advances without premature discarding of events.

4.2 DataStream API

The design and implementation of our system is done via Flink's Datastream API. The base object for manipulating streaming data is a DataStream which can be initialized by specifying a source of data (e.g. a file). After the creation of a DataStream, users can then manipulate the data using several operators, some of which are described in this section due to their relevance in the designing of the application. Operators represent the operational nodes that are used to create the job graphs, which in turn allows the computations to be distributed.

Map: This operator allows the user to specify a function that takes one element and produces one element by applying a transformation function. For instance, it could receive a stream of integers and return their value multiplied by two. This function is used when each element in the stream must be submitted to the same transformations under the same conditions.

FlatMap: This operator allows the user to specify a function that takes one element and produces zero or more elements. For instance, a user could specify a function that divides a sentence into words. In this case, one event (sentence) could produce several others (words). This function is more flexible than the map function, as such, it can be used in a wider range of scenarios.

Connect: Connects two data streams (possibly coming from different sources) while retaining their types. Connecting two streams allows users to manipulate data from both concurrently in the same operator (The CoFlatMap). Since they will be available inside the same operator, both streams share the same state.

CoFlatMap: The CoFlatMap is a special function that allows a user to process two distinct streams simultaneously after they have been connected by the Connect operator. In a FlatMap operator, the incoming all the events in the stream must belong to the same type of object (e.g. String). The CoFlatMap allows streams of different types (e.g. String objects and Rule Objects) to be processed in the same operational node while sharing the same state.

Filter: This operator allows the user to specify a filter function to be matched against each incoming element. If the function returns true the element is forwarded to the next operator or discarded otherwise.

KeyBy: Splits a stream into disjoint buckets. For instance, if we have an incoming stream of network packets and if the specified key is source address, new buckets will be created, each one containing only packets for a specific source address. A bucket is an isolated container that holds only events of a specified key.

Window: Windows group the data according to some characteristics. In Flink, there are Tumbling windows, Count windows and Sliding windows. For instance, after defining the key source address one could define a tumbling window. This would mean that for each distinct group of IP addresses a new window would be created as the example in Figure 4.5.

Apply: This function is applied to a collection of elements. It Iterates through the elements in the window applying some specific aggregation function (e.g. count).

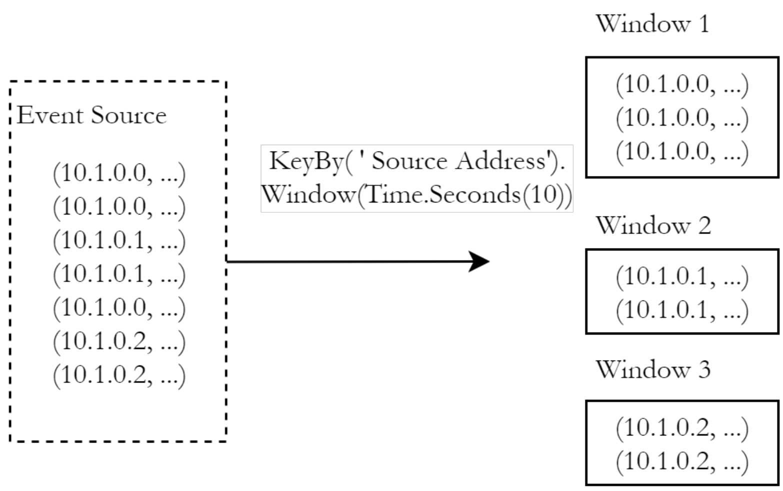


Figure 4.5: Flink window function

AssignTimestamps : Both timestamps and watermarks in Flink are long values specified in milliseconds. Since our events can have several formats for the timestamp, it is important to correctly convert these values. This conversion and assignment is done in the Timestamps assigner function.

Split: This is used to split a stream into several streams using some criterion. For instance, one could split a stream of integers into "odd" if $integer\%2! = 0$ or "even" otherwise.

Select: This is used to select streams after they have been split by the split operator.

There is a particular problem with the $KeyBy$ operator that may potentiality create a bottleneck in the stream. For instance, assuming that a stream of events that have a field Key whose values can be A, B, C or D. When a user applies the operation $KeyBy('Key')$ the incoming elements can form four groups (A, B, C or D). These grouped elements are then sent, according to their key, to other nodes for execution. The values corresponding to the same key must be present on the same node so that the followed functions are accurately calculated (with no missing element). The problem is that if one key is dominant (it appears more often) the node responsible for applying the functions to that key will have to process more elements. Because Flink does not allow the users to choose which machine will be responsible for performing each operation a dominant key can easily cause a bottleneck. The Figure 4.6 illustrates an example where we have two partitions that will hold the buckets for each key and where A is the dominant key (present on the partition 1).

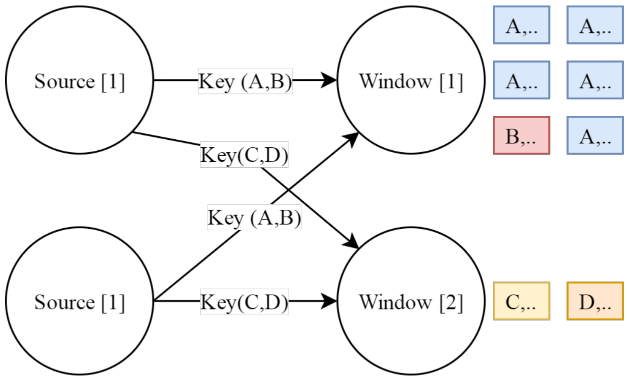


Figure 4.6: Dominant key problem

4.2.1 Anatomy of a Flink Program

When a user creates a program using one of the provided APIs, he is essentially creating an execution plan that specifies how the data should be collected, transformed and sent from operator to operator until it reaches a sink. In order to explain the anatomy of a Flink program, we will give an example of a typical application called word count. The objective is to count the number of equal words in an incoming stream of Strings.

The Stream Execution Environment is the basis for all streaming applications in Flink. The execution environment provides methods to control the execution of the program. For instance, the user can manipulate the frequency of checkpoints and the global parallelism of each operation. It is also the starting point to interact with the outside world in order to collect data. The example below depicts a source node that will collect the data from a file.

```
final StreamExecutionEnvironment environment =
    StreamExecutionEnvironment.getExecutionEnvironment();
DataStream<String> text = environment.readTextFile("file:///path/to/
    file");
```

Listing 4.1: Flink Execution Environment and Source

The *DataStream* represents the abstraction which contains the streaming data and allows its manipulation. The DataStream can be manipulated to create new data streams by applying transformations on data. For instance, one could apply a FlatMapFunction to transform the Strings, collected from the file, into word pairs by splitting the String between spaces.

```
text.flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>() {
  @Override
  public void flatMap(String sentence, Collector<Tuple2<String,Integer
    >> words) throws Exception {
                    for(String word: sentence.split(" ")){
                        words.collect(new Tuple2<>(word,1));
                    }
                } });
```

Listing 4.2: Words Splitter

After splitting each sentence into pairs (word, value), the only thing left is to group the matching words and sum their values. for instance, if we have the following lines,

”it is amazing
so amazing”

each word can be grouped as illustrated in figure 4.7.

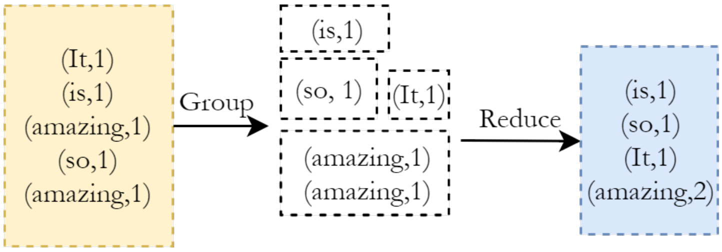


Figure 4.7: Word count - group and reduce

Ultimately, to replicate this behavior in a Flink program, one simply needs to group elements by word (the first element in the tuple) and sum the values (the second element in the tuple). The listing below illustrates the full program to count words in sentences.

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.
    getExecutionEnvironment();

DataStream<String> text = env.readTextFile("file:///path/to/file");
DataStream<Tuple2<String, Integer>> dataStream = text
                    .flatMap(new Splitter())
                    .keyBy(0) //Key by the String
                    .sum(1); // Sum the integer values
dataStream.print();
env.execute("WordCount");
```

Listing 4.3: Words Splitter

The sample programmed created above uses the default behavior associated with Flink’s operators. The implementation of our system, which we will describe in the next chapter, uses custom implementations of these operations to allow different kinds of behaviors, particular to our needs.

Chapter 5

Design and Implementation

This chapter describes the design and implementation of our system in a top-down approach. Primarily, in Section 5.1 we provide a high-level description of the system's architecture and its main components. After the initial description, we give a detailed explanation of each component, namely the EPL in Section 5.2, which allows users to interact with the system, the Storage Layer in Section 5.3, which stores part of the state that is used to process the events, the transport layer in Section 5.4, to where logs are published to be later consumed and, finally, the Correlation Engine in Section 5.5 which is the main component of the solution that was developed using Flink.

5.1 Architecture

The system architecture can be divided into four main components: the interface, the storage layer, the transport layer and the correlation engine. Figure 5.1 illustrates the interaction between each of those components.

We want to allow users to insert, edit or delete rules as needed. To serve this purpose, we created our own EPL and embedded it in a simple interface, which was created using the Netbeans Platform[38].This platform allows the development of custom applications using a generic framework that contains several GUI components and other optional modules. We will not delve into detailed explanations about this components as it falls outside the scope of this work. However, understanding how does this layer interact with the rest of the components is crucial to the comprehension of the complete life-cycle of a rule. All the rules created using the interface are uploaded to a Git repository [39], which is capable of maintaining a history of changes and their associated authors. Succinctly, the rules contain a description of the necessary conditions to filter events and extract relevant information from the gathered data.

Besides the Git repository, which holds the rules, the storage layer also contains a database that has IOCs and custom tables. Each IOC table represents a set of malicious artifacts of the same type (e.g. Malicious URLs) that may be populated by other systems

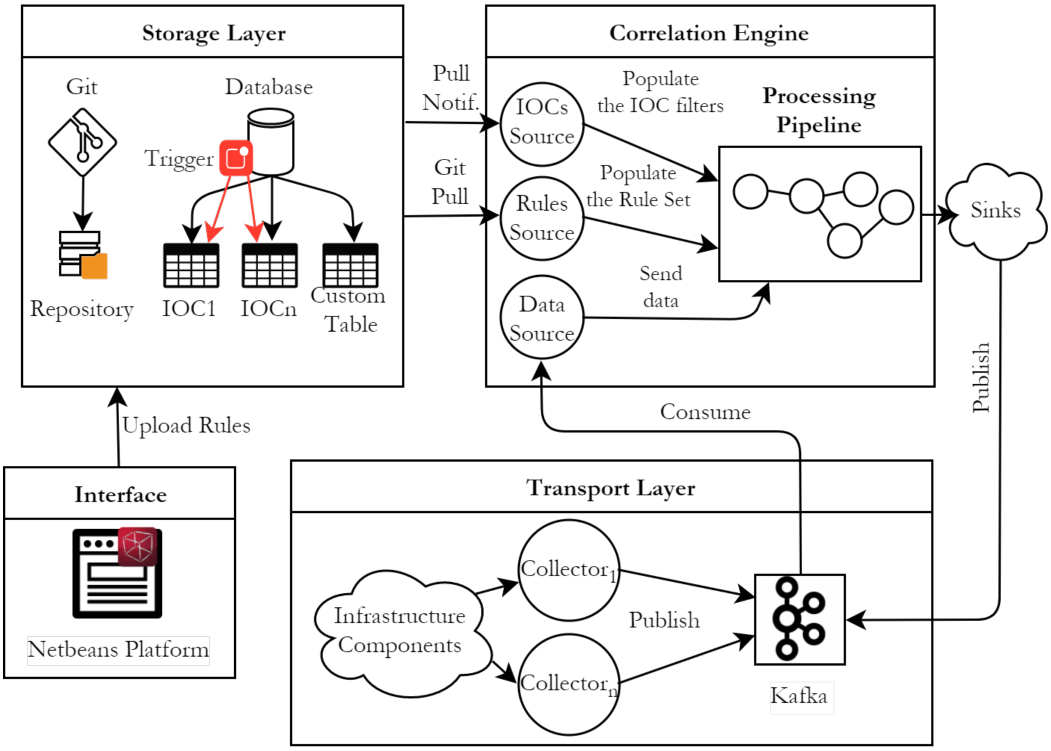


Figure 5.1: System architecture

or by users. The custom tables are used as auxiliary data in the development of rules. These tables can be arbitrarily populated by users and queried using rule clauses to match one or more event fields to one or more tables. Because the IOC lists can have millions of entries, directly querying those tables from the correlation engine would considerably increase the amount of time it takes to process an event. Therefore, each IOC list has an associated trigger that is responsible for detecting changes, namely insertions, deletions, and updates. The IOC source node of the correlation engine regularly pulls these changes and applies them to a collection of Cuckoo filters. Querying these cuckoo filters is several orders of magnitude faster, therefore they present a major advantage.

The transport layer consists of a message broker, in this case, Kafka, to where logs are published by event collectors and consumed by the Correlation Engine to be analyzed by the rules.The broker can have several partitions, which allow the published logs to be consumed in parallel by the parallel instances of the Correlation Engine.

Lastly, the Correlation Engine is the core component of the architecture, which represents the distributed CEP system that we created using Flink. Essentially, the Correlation Engine is a chain of nodes (or a pipeline) that is used to analyze an event in a series of iterations. In Figure 5.1 the sources nodes are segregated from the pipeline to illustrate that the engine will gather data from different sources, nevertheless they are still it is part of the chain. Each node in the chain was programmed using the Flink Streaming API and has a specific role with one primary goal - to dynamically apply the rules to the incoming events. The source nodes (e.g. data source node) collect data from outside. Filter nodes, apply filter functions to the events coming from the source nodes and discard them if they do not match any filter or forward them otherwise. Aggregation nodes are used to aggregate events into windows and apply aggregation functions. Lastly, sink nodes are used to send the results to the respective destinations. The results of the correlation can be published to a persistent storage and/or other third party software to be further analyzed. Additionally, the results are also reinserted into the engine to be correlated with other incoming data.

5.2 Event Processing Language

Instead of using an already existing language we developed our own EPL both because we needed the language expressions to match Flink function arguments and also because it enabled us to create custom expressions tailored to our needs and similar to already existing constructs in commercial products.

The language specification was written using ANTLR (Another tool for language recognition) [40], which is a framework that allows the construction of interpreters and compilers. By defining a language specification in ANTLR it can automatically generate a parser file to be used in a specific language such as Java or Python. A language specification is given by two files, the lexer where the tokens are defined (i.e. the clauses that make-up the language) and the parser which specifies what constructions are allowed. A grammar file can also be used which can contain both tokens and the language semantics. From our grammar, which can be found on Appendix B, we generated a parser using ANTLR and created a custom listener which extracts the rule parameters to populate a Java Object. This object can then be stored in a set of rules of the correlation engine, which will become more clear in Section 5.5. Additionally, the parser was also used in our interface for syntax coloring and error highlighting. Figure 5.2 shows the possible clauses that can be used to create a rule using our EPL.

The rule clause is used to associate a unique name to the rule which is used as an identification tag in the Correlation Engine. The source clause is used to identify if a rule belongs (or not) to a certain source (e.g. *source field = firewall*). The Filter clause has a function that is used to filter out (i.e. discard) the events that do not match the specified conditions. This function must be written in a language that allows the script be executed at runtime, in our case we use JavaScript. Instead of having a JavaScript in the Filter clause we could extend the set of possible clauses so that users could specify filtering conditions using those clauses. This approach, however, would limit the possibilities that users have to manipulate event fields as in an expressive language such as JavaScript and would increase the complexity of the correlation engine, which would have to be able to handle all the distinct input variations. This design decision will become more clear

RULE *string_literal* SOURCES *identifier = [identifier (, identifier)*]* FILTER *function* [IOC FILTERS ((*ioc* | *not_in_ioc*) ioc (*event_header*))*]	*Filter Clauses*
[USING *identifier* AGGREGATE BY *identifier (, identifier)** WITHIN *length*,*interval* [SORT BY *identifier*] HAVING *function*]	*Aggregation Clauses*
SINK *function*	*Sink Clause*

Figure 5.2: EPL schema

in Section 5.5. The IOC filter clause is used to match events against the entries in IOC lists. If the specified fields match the IOC lists, the event is forwarded, otherwise, it is discarded. Events for rules that do not specify this clause are simply sent to the next operator.

Rules are subdivided into two types, Aggregation Rules, which use the optional aggregation clauses and Simple Rules that are directly sent to a sink if the event matches the filter function. We divided the rules into two types because events are treated differently according to the rule type they fit in. Simple rules, completely ignore the timestamps of events as the conditions do not depend on them. Contrarily, Aggregation Rules store events in time windows and, as such, they must inspect the event timestamps to be able to group them correctly. In Aggregation Rules the Using clause is used to define which event field contains the timestamp that should be used to create windows. The Aggregate By clause defines the keys to be used when grouping events. Events with equal key values are stored in the same group. The Within parameters are used to create windows in Flink according to the specified length and interval. The Sort By is used to optionally sort events according to a specific field before applying the aggregation function. The having clause is the aggregation function, also written in javascript, to be applied to the set of grouped events for every window interval. Ultimately, the Sink function is used to prepare the event before sending it to the desired destinations.

The automaton for the language is represented in Figure 5.3. By inspecting the automaton one can easily see dependencies between different clauses and how the rule schema functions as a whole.

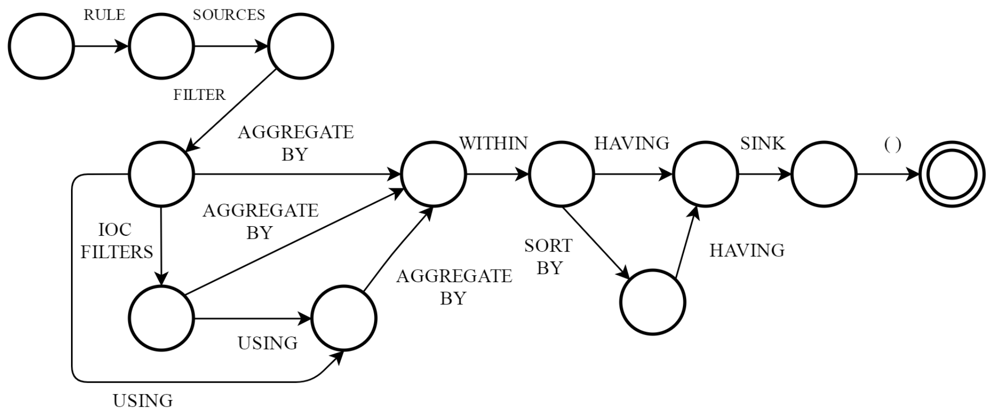


Figure 5.3: EPL automaton

5.3 Storage Layer

The main purpose of the storage layer is to persistently store rules, IOCs, and custom tables as well as to allow their correct management without directly interfering with the Correlation Engine. All of our rules are stored in a Git repository. Git is able to maintain a history of changes to the repository, which not only can be associated with a specific user but also allow us to revert a rule to a previous state if something goes wrong. Our rules source node of the correlation engine regularly pulls data from the Git repository according to a specified interval. All the changes that are observed between two pulls are applied to the internal state, for instance, if a rule is deleted from the repository then it is removed from the internal set rules. This set of rules, as well as their enforcement, is explained in later sections.

For performance reasons, all the entries for IOC tables are used to populate a set of Cuckoo filters that is maintained by the IOC filter nodes of the correlation engine. For that, we created a trigger function that we associated with each IOC table. When an IOC table is modified, the trigger creates a new notification that contains the action (e.g. Insert) and the associated row. In turn, the Correlation Engine has an IOC source node that periodically fetches the notifications, similarly to the rules source node, and inspects them in order to determine what action to take. For instance, if the event action is the insertion of a new row in a specific IOC table than that entry is also added to the respective Cuckoo filter, if it exists, otherwise, a new cuckoo filter is created and the entry added afterwards as Figure 5.4 shows. The Cuckoo filters can instead be queried using the IOC FILTERS clause.

Custom lists are created as needed by the users for auxiliary purposes. For instance, a user may want to create a filter expression that matches a set of integers. For that, he could create a database table and query it instead of manually verifying if the event fields match those integers. This mechanism is useful because if the integer set changes instead of changing all the rules which use that particular filtering expression the user simply

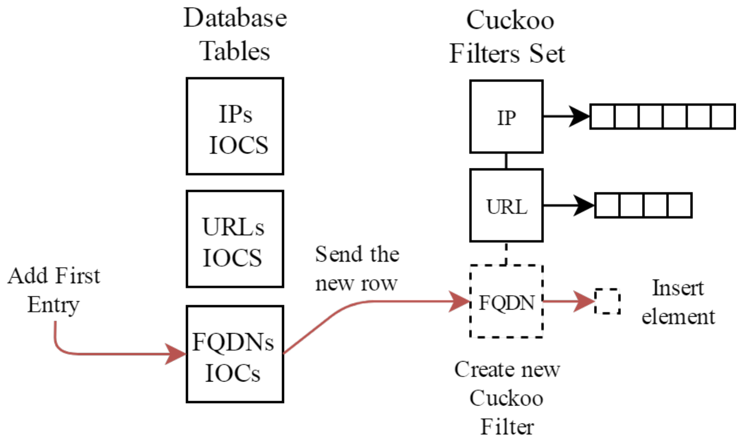


Figure 5.4: Creation and population of Cuckoo Filters

needs to change the associated table and the rules that query that table will automatically have the new changed set.

Lastly, the Correlation engine also has the ability to change the database tables. For instance, the results of a rule may be used to populate a table, this can be done in the sink function of a rule. Another important change that the Correlation Engine can make is to deactivate a rule. When a rule is deactivated, it is removed from the engine rule set. A rule can be deactivated due to several reasons. To name a few, the rule may be consuming too much CPU time or its memory usage is higher than it should be, hence it needs to be deactivated. This feature is part of a protection mechanism that is discussed latter in Chapter 6.

5.4 Transport Layer

The transport layer is composed by event collectors and a message broker. Event collectors are external components that retrieve the logs from the infrastructure components using a pull or push strategy. After collecting the raw logs, the collectors parse them to a common format before sending them to Kafka. The message broker is constantly appending the parsed logs to the partitions of a specific topic. The data source node of the Correlation engine, in turn, continuously fetches the logs and forwards them to the filter nodes.

Instead of storing all events in the same partition, the Kafka cluster maintains a partitioned log where each partition is an ordered, immutable sequence of events that can be continuously appended to. Partitions allow the logs to be scattered across different servers and can act as a unit of parallelism. The data source nodes can be subdivided into several parallel instances (execution nodes) that consume events in parallel from each of the partitions increasing the overall throughput, as Figure 5.5 shows.

Since the correlation engine also publishes its results to Kafka, it acts as a ”natural”

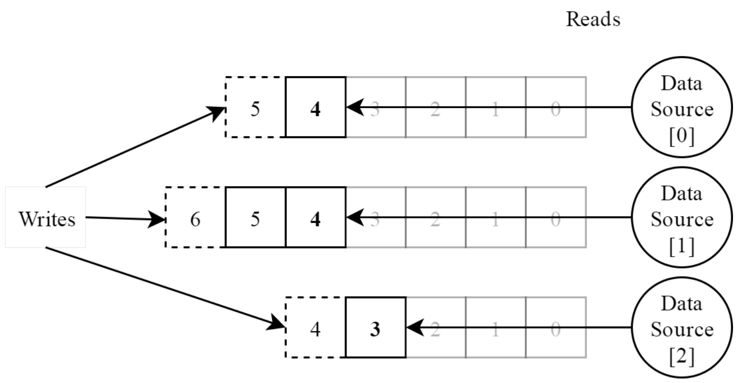


Figure 5.5: Kafka parallel consumption

reinsertion mechanism because the events published are consumed again by the engine to be correlated with new ones. Additionally, since Kafka retains the logs they can easily be consumed by other applications besides the correlation engine without any interference.

5.5 Correlation Engine

The Correlation Engine is the brain of the operation and contains all the mechanisms that can be used to detect security threats. All the events consumed from Kafka are processed in a series of chained operations which constitute our Processing Pipeline. The pipeline is essentially a Flink Job Graph that allows the extraction of information from events in a series of steps and that is designed to be easily scalable as our necessities grow. In the following subsections, we describe the mechanisms that we implemented to allow rules to be dynamically stored and applied to the stream of events without having to start and stop the Flink job. Additionally, we also describe the role of each operational node that constitutes the pipeline and what was the reasoning behind some design decisions.

5.5.1 Dynamic Rules

As mentioned in the previous chapter, Flink's execution plan cannot be changed dynamically (i.e. at runtime). Therefore we need to add new functionality that allows us to add, remove or delete rules at runtime to satisfy our requirements. A typical Flink application (or Job) will have a set of operational nodes that work together to continuously apply transformations to a datastream and output some results. These transformations are static by default, meaning that a node will always apply the same function to each event. In order to change the behavior of the application the developer as to reprogram the operational node(s), compile a new job, stop the old one and submit the new. Assuming that we have a stream of events with two fields: an integer and a string. Initially we could create a function to multiply every integer by two when the string is *A001*. Latter, if we wanted to add a new condition to multiply by three events with the string *A002*, we would have to reprogram the job, specifying this new condition, stop the currently running job and submit a new one, as Figure 5.6.

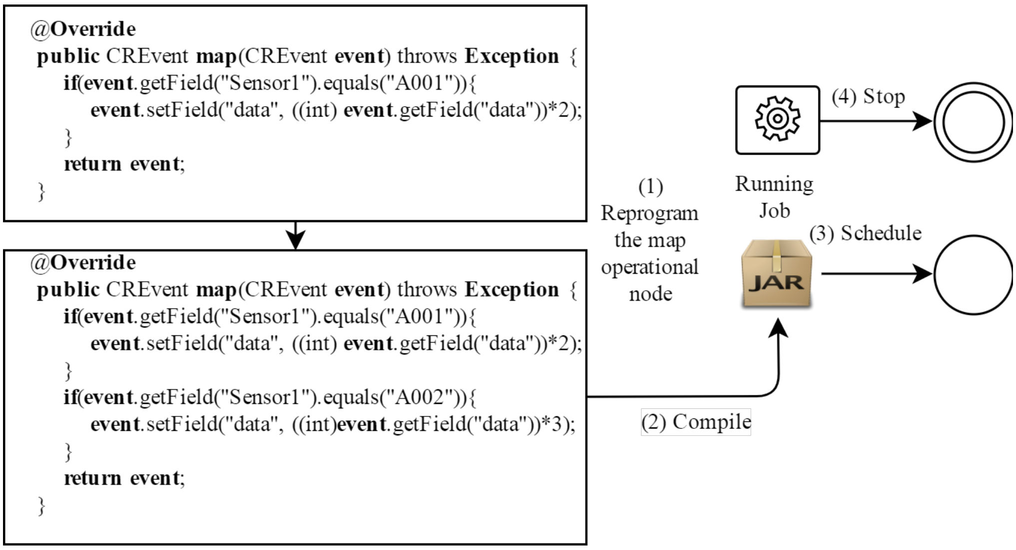


Figure 5.6: Job Scheduling

For our particular use case, this mechanism does not suffice. To begin with, our existing set of security rules are constantly changing meaning that we would regularly have to modify the job by adding new conditions. This would quickly make the application very extensive and thus more error prone and harder to debug. Furthermore, by having to stop and start the job for each change we would lose system availability, which is not desired. Alternately, we could deploy a new job while the old is still running and only killing it when the newly deployed job is ready. However, in order to do this we would need to double the computational power to allow both jobs to run simultaneously. Moreover, it would make the code even more complex and error prone.

Another option is to deploy an individual job for each security rule and maintain it independently. Although this method would make each individual job very simple, it would create two new problems. First, we would have to monitor each job independently, which is very time-consuming and would imply a lot of operational work in our daily activities. Secondly, since some rules are very similar and maybe interested in the same set of events, we would have a lot of replicated data between each of these jobs, increasing the hardware requirements.

In order to overcome this problem, we followed a similar approach to King's [41] solution. King is a video game company with more than 300 million unique users and over 30 billion events coming from different games and systems every day. To allow data scientists to efficiently gather important information about their games, they decided to develop their own ESP system using Flink. One of their goals is to provide an easy interface for data scientists to create new analysis modules without having to worry about Flink constructs. This is a very important aspect since data scientists are very good at analyzing data but they often lack advanced programming skills. Furthermore, in order to make debugging easier and to avoid users having to submit a new job every time they need to analyze the data, their application supports the creation of new modules dynamically. Since we also want to support this functionality, our solution closely follows some of their design ideas. However their particular implementation details or documentation were not made available, this means that even when our approach to solving a particular problem is the same, the mechanisms used will naturally differ.

In order to dynamically apply new logic to incoming events, King's solution consists in receiving two simultaneous streams, one stream of events and one stream of scripts in a single operational node. The stream of events represents video game metadata while the stream of scripts contains the logic that must be applied to those events. Since the running job code cannot be changed at runtime, they solved the problem by having a stream of scripts written in the Groovy language [42] that can be compiled at runtime. Groovy can easily be integrated with Java, where the programmer has the possibility of adding new variables and new functions to the execution environment, thus changing the running job. The groovy functions can manipulate the stored data and output new results. Using this approach, the scripts coming from the stream can be easily compiled and stored. Every time a new event arrives from the stream of events, its data can be stored in the Groovy execution environment so that the functions can process that event before forwarding it to the next operators. Although Groovy scripts are a powerful tool, one cannot run Flink operations inside them. Instead, the scripts associate new metadata to each incoming event so that events can be processed using the correct Flink operators, which can be programmed to extract the metadata. Furthermore, since the scripts execution is much slower than directly executing the operations in Java, running heavier computations on these reduces the overall performance.

In order to support this functionality, Flink's job graph as to be prepared to receive both events and scripts in the same operator and configured to interpret the event metadata in order to apply the desired functions. Figure 5.7 shows how would one configure the job graph in order to support such functionality.

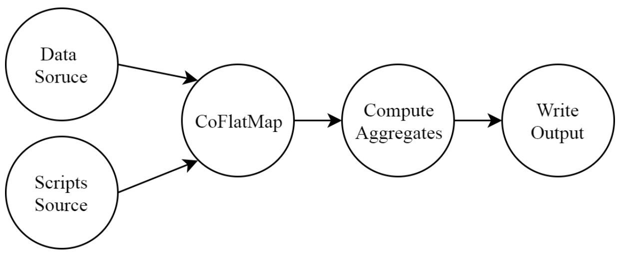


Figure 5.7: King's pipeline

To begin with, the job has to have two source nodes, one that gathers event data (games metadata) and the other that receives the scripts. After that, to process both scripts and events in the same operation one must use a *CoFlatMapFunction* as explained in the previous chapter. Using this function, both scripts and events will have access to the same state, which allows the stored scripts to have access to the events. With both scripts and events in the same operator, every time a new event arrives it can be processed by every script which creates a new copy of the event with its specific metadata and forwards it to the compute aggregates node. This metadata includes a window length and the aggregation function that needs to be applied. When an event arrives at the aggregates node, it inspects the event timestamp t and the metadata containing the window length l. After that, the operator stores the event in a window starting at t and ending at $t + l$. Every next event whose timestamp is lesser than $t + l$ and larger than t would be stored in the same window. After creating a window of events, the desired aggregation function, specified by the event metadata can be applied. Lastly, the results are written to the desired destinations.

As an illustrative example, let us assume that a user creates two scripts: one to count the number of events that represent a game end and another to calculate the average time a user takes to complete a game, both using a one-minute windows. Additionally, each event has a field that indicates the type of event, for instance, if it is a game start or a game. Events that represent a game end contain the duration of the game. Both our scripts will only match events which are games ends. For this, the user would create two scripts, one where he would specify that all the events should be aggregated in a one-minute window and counted and another similar one but instead of counting the number of events it calculates the average time spent.

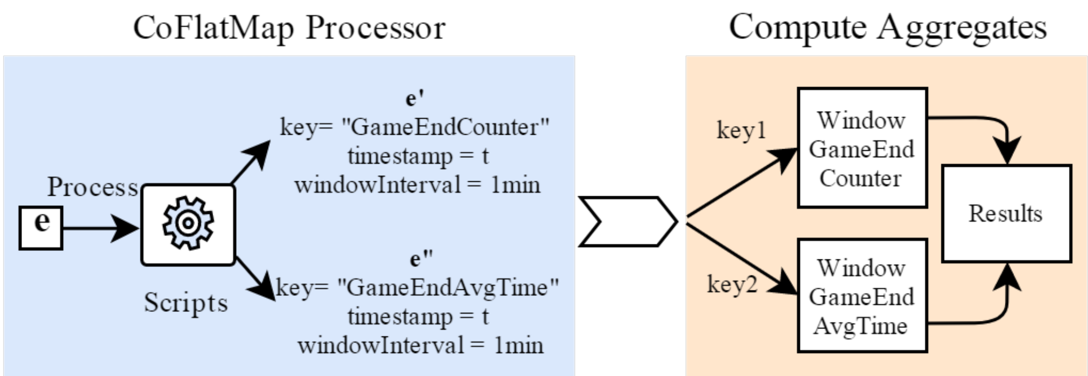


Figure 5.8: King's dynamic scripts

As we can see in figure 5.8, when a new event arrives at the system it is processed by both scripts that output two different events with distinct keys, the gameEndCounter associated with the first script and the gameEndAvgTime with the second. When the generated events arrive at the compute aggregates they are validated and their metadata is used to create two distinct windows both which aim to aggregate events in a 1 minute time window but for distinct keys. Every new event will have to be associated with the respective window. When it has gathered one minute's worth of events, the aggregation function, given by the associated metadata, is applied. Grouping elements and assigning windows to them is made by Flink's *KeyBy* and *WindowAssigner* operators discussed in the previous chapter.

Using some of the concepts from the described approach, we developed our own solution which is described in the following section.

5.5.2 Processing Pipeline

Similarly to the King's approach, we also have a stream of scripts and a stream of events. The scripts represent our security rules and the events the logs of the monitored components. Instead of having the scripts written in Groovy we developed our own EPL that supports embedded JavaScript functions as described previously. In order to correctly parse the embedded JavaScript functions, we combined our grammar with an ECMA (European Computer Manufacturers Association) scripting language grammar. The ECMAScript gave birth to languages such as JavaScript and JScript which share the same basic structure. The grammar [43] was developed by Bart Kiers at MIT.

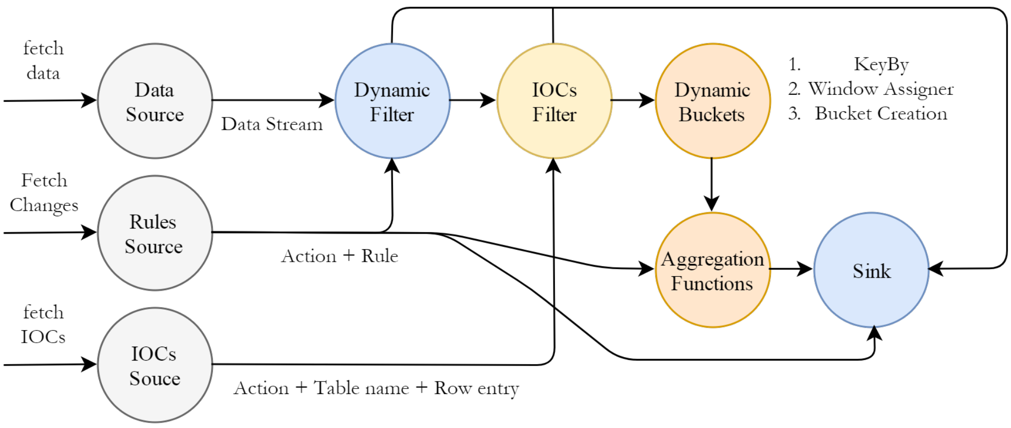


Figure 5.9: Processing pipeline

Figure 5.9 shows our pipeline that was configured, similarly to king's solution, to receive both scripts and events. When the rules source node fetches a new set of changes from the Git repository it associates each change with an action, which can be a deletion, edition or creation of a rule, according to the action that was taken by the author of the change. Then it sends a new rule event to the filter node that contains the action and the textual representation of the rule. The filter node is a *CoFlatMap* implementation which receives both rules and events. When it receives a rule if the action is a delete, the rule is removed from the set of rules. If the action is a create, then the rule is added to the set of rules. An edit is similar to a create but in this case, the old rule is replaced with the new one. Before storing a rule, the filter node parses it and stores the JavaScript filter function and the rule name in a java object. All the rules are stored in a tree-like structure that we developed named rules-tree. This structure is designed so that an event quickly matches a rule that has its source key and value.

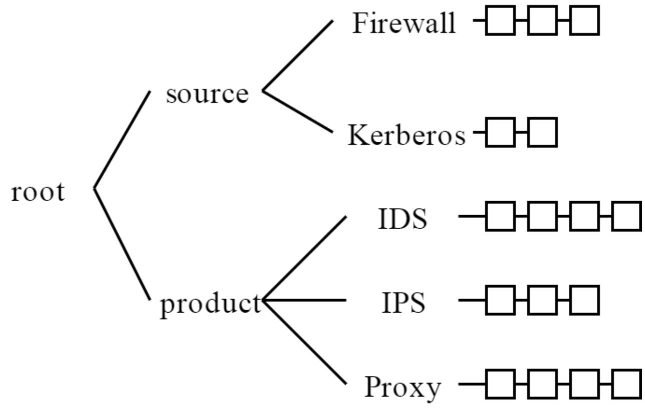


Figure 5.10: Rules Tree Strucutre

Figure 5.10 shows an example of a tree which has five different sources, each one a different set of rules. When a user creates a rule he has to specify, in the sources clause, what are the event sources that he is interested in. As an example, one could set the source as *SOURCES product = ['IDS','IPS']*. This means that the rule would only accept events which have a field called *product* with the values *IDS* or *IPS*. Therefore an event with the *product* key and either IDS or IPS as a value would only have to be processed by their set of rules. This strategy considerable reduces the time it takes to process an event since it only has to be matched against a smaller set of rules. This structure was constructed using a Concurrent HashMap as the primary tree whose keys are the source keys and values another HashMap that contains as keys the sources values and as values the set of rules. Using a Concurrent HashMap the rules and events can be processed simultaneously assuring consistency at the cost of losing some performance when a rule is being added or deleted.

When an event arrives at the data source node, it is parsed to a map which holds the event fields and their respective values, before being forwarded to the Dynamic Filter node. Each event that arrives at the Dynamic Filter node is processed by the respective set of stored rules. First, the rules that have the same key-value sources as the event are retrieved from the rule-tree. If the event has the field *source="firewall"* then the firewall rule set is retrieved, if it exists. After that, the event is processed by the JavaScript filter function of each retrieved rule. For each rule whose filter function returns true, the event is tagged with that rule's name and forwarded. Additionally, if the rule has aggregation clause, the event is also tagged with the window length, window interval and the aggregation keys. If the event does not match any rule (i.e. the filter function returns false in all cases) it is discarded. Supposing that the event has a field named *id* whose value is 3 and that it will be processed by the following two filter functions.

```
//Other Clauses omitted for simplicity.
Rule "sample 1"
FILTER function first(event){
  return event.id == 7 ;
}
Rule "sample 2"
FILTER function second(event){
  return event.id == 3;
}
```

Listing 5.1: Sample Filter Function Clauses

Since the event id is *3* the first function returns false. As for the second function it returns true and, therefore, the event is tagged with the rule named*sample 2* and forwarded for further processing. Events are forwarded to different nodes according to the type of associated rule. If the rule only has a filter function followed by the mandatory sink function, it is considered a simple rule. Therefore after being processed by the filter node it is sent to the sink node. A rule that has an IOC filter clause in addition to the filter clause is sent to the IOC filter to be matched against the IOC lists. Lastly, if the event is associated with an aggregation rule, it is sent to the Dynamic Buckets node. Due to the large amount of events the filter node has to process, as well as due to the complexity of the operations and the time it takes to execute the JavaScript functions, it may potentially cause a bottleneck. As such, the parallelism level should be the highest of the pipeline so that each spawned node can process a smaller set of events since these are distributed in a round-robin fashion to every execution node.

The IOC filter contains our specific *CoFlatMap* implementation, which in this case receives both events from the filter node and IOCs from the IOCs source node. Due to a Flink limitation, only two streams can share the same state. Therefore the IOC filter node cannot simultaneously receive rules, events and IOCs. Instead, in the filter node when a rule has the IOC filter clause the event is also tagged with the IOC FILTERS clause arguments. For instance, if the IOC filter is *IOC FILTERS in ioc ipAddresses(sourceAddress)* then the node should check if the field *sourceAddress* of the event belongs to the cuckoo filter *ipAddresses*. If the event belongs to the specified lists it is forwarded to the Bucket Creation node, otherwise, it is discarded.

The Bucket creation node has three custom chained operations: the key by, the window assigner and the bucket creation. The key by operator splits the events according to their aggregation keys, by inspecting their *aggregate by* tag. Afterwards, events with equal keys that belong to the same time window are grouped. Lastly, the bucket creation node will join all of the events that belong to the same window and forward them to the aggregation functions node.

The aggregation node, similarly to the filter node is a *CoFlatMap* custom implementation that receives both rules and events. This time, the rules are simply stored in a concurrent hash map whose key is the rule name and the value the compiled JavaScript aggregation function. When it receives a bucket, the aggregation node inspects the rule that is associated with the events from that bucket and retrieves it from the rule set. Afterwards, the aggregation function is applied resulting in a complex event that is sent to the sink node.

The sink node is the last piece of the pipeline. It gathers the events that were created as a result of simple and aggregation rules and processes them. The sink node also has a HashMap of functions similar to the aggregation functions node that it uses to retrieve the rule associated with the complex event and apply the respective JavaScript Sink function, which prepares the event before it is sent to the destinations.

We intended to make this pipeline as bottleneck-free as possible by segregating the events and only forwarding them to the nodes of interest. The same can be said for rules because each clause of the rule is only forwarded to the node of interest, the filter function to the filter node, the aggregation clauses to the bucket creation node and so on. Moreover, separating the operational nodes this way allows us to easily identify the operations that need more computational power and thus need to have a higher level of parallelism.

As an illustrative example, let us assume that a user creates the following security rule.

```
RULE "Several login attempts with non existing accounts"
        SOURCES product = "Kerberos Server"
        FILTER function(event){
                return utils.includesIgnoreCase(event['message'],
                                'The specified user does not exist.');
                }
        AGGREGATE BY UserName, destinationAddress
        WITHIN 1:min,1:min
        HAVING function(events){
                return events.length >= 10;
        }
        SINK function(event){
            event['message'] = 'Several login attempts with non
    existing accounts';
                return event;
        }
```

Listing 5.2: Rule Sample - Several login attempts with non existing accounts

This sample rule is supposed to trigger an alert if a certain user tries to log-in more than 10 times in a minute to a specific server with a non-existent account. After its creation, the rule is stored as part of the state of the application as described previously. When an event arrives, the filter function is applied and if the event contains the specified message in the *message* field it is tagged with that rule's name, the aggregation keys, the window length and the window interval. In the bucket creation phase, Flink will split the events according to their specified aggregation keys (user and destination address) and store them in the respective windows. As soon as the window closes the bucket is sent to the aggregation functions node, which inspects the events and retrieves the compiled JavaScript function of their associated rule. In this case, it is a simple count that forwards an alert to a Flink sink if the number of events is equal or larger than 10.

Dealing with late Events

As discussed in the previous chapter, windows are created and closed based on watermarks. Since watermarks are emitted according to the observed event timestamps, if these are not correct or if events arrive out of order, windows can be closed prematurely (late events belonging to that window may not yet have arrived). When the window is closed, the aggregation function is applied and, after that, the events belonging to that window are discarded. Due to network latency, processing time and other factors, an event can arrive after the end-time of a window to which it belongs. Therefore, we must employ a mechanism that deals with these events and associates them with the correct window. Fortunately, Flink already has a way to deal with this problem by allowing the specification of an allowed lateness interval in the window operator by adding an additional parameter. All the events that belong to a window aggregation are stored in memory for the specified lateness interval, all the events that arrive within this period will trigger the aggregation function. The problem is that the larger the lateness interval the higher the memory consumption, hence we cannot set an arbitrarily large interval.

As discussed in the previous chapter, Flink employs a watermarking mechanism to advance time. Instead of constantly increasing the watermark value we return watermarks as the current highest observed timestamp minus an out of orderness bound. This bound is used to account for network latency variations and other factors so that a window is not closed too soon.

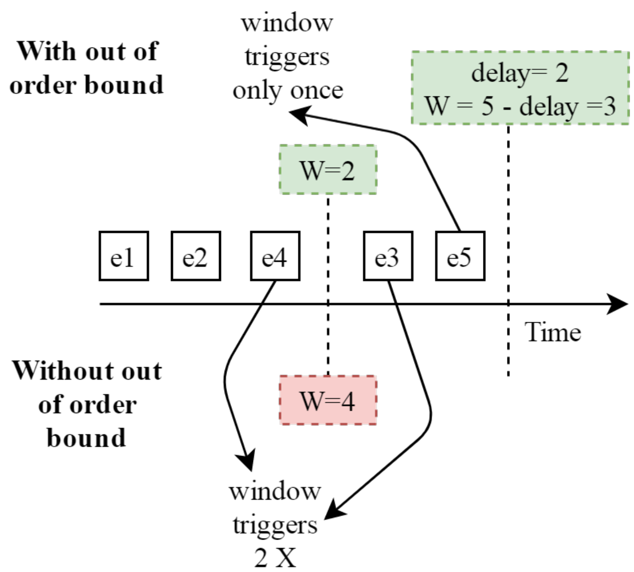


Figure 5.11: Out of orderness bound

As an example, in Figure 5.11 we have five events ($e_1 ... e_5$) that arrive in the illustrated order and whose numbers represent their respective timestamps. Additionally, let us assume that the events are to be grouped in a window whose start-time is 1 and end-time is 3. If we only set the allowed lateness, as soon as the event e_4 arrives the watermark advances to 4 and the window is closed. This gives us an undesirable result (element e_3 is missing). When the event e_3 finally arrives, the window function is triggered again ($e3$ belongs to the stored aggregated elements). Because of this, we would have two results: an earlier incomplete result and a correct one. In this case, by setting the out-of-orderness bound to 2, when event e_4 arrives, the window will not be triggered because the watermark will only advance to 2 $(4 - 2)$. Finally, when event e_5 arrives the watermark value advances to 3, the window closes and we only see the correct computed value.

Besides this mechanism, all the events are stored for a period of time defined by Flink's allowed lateness. No guarantees are given for events that arrive beyond the lateness interval.

5.5.3 Indicators of compromise

As mentioned in previous sections, we maintain lists of IOCs as database tables that can be manipulated by end-users. For our particular use case, these IOC lists can have millions of entries, therefore a query to match an event field to a database table entry can take several seconds. This latency can not only significantly delay the detection of an attack but also cause a bottleneck in the processing pipeline. As a first approach, we could try to store the IOC lists in the correlation engine in a commonly used data-structure like an HashMap. The problem with this approach, however, is the high memory consumption that would require several hundreds of gigabytes of RAM, which makes the solution infeasible. In order to overcome this problem we decided to use the Cuckoo Filter as described in Section 2.5 of Chapter 2. There are several reasons which made us decide to use this data-structure. First and most obvious is that the size occupancy can be several orders of magnitude lower than commonly used data-structures. Secondly, since false negatives do not occur, an attack that could be identified by an IOC list would never go unseen. As for the false positive probability, we can set its value to a rate which we are comfortable with and that does not increase the size occupancy beyond our memory capacity. Third, the Cuckoo Filter allows the deletion of entries, which is useful for deleting an entry when the IOC is removed from the database, without having to rebuild the filter. Lastly, the Cuckoo Filter can be shared with other entities without having to disclose the underlying elements.

To effectively use these probabilistic filters in our processing pipeline we stored them in our IOCs Filter node, as mention in previous sections. Every event that arrives at the IOC filter node will have the associated the IOC filter clause arguments as a set of key-value pairs. The key represents the IOC table, which matches the Cuckoo filter name and the value the event field that should be used to see if the event belongs to the filter. Figure 5.12 depicts the process used to query the Cuckoo filters on the IOC Filter node.

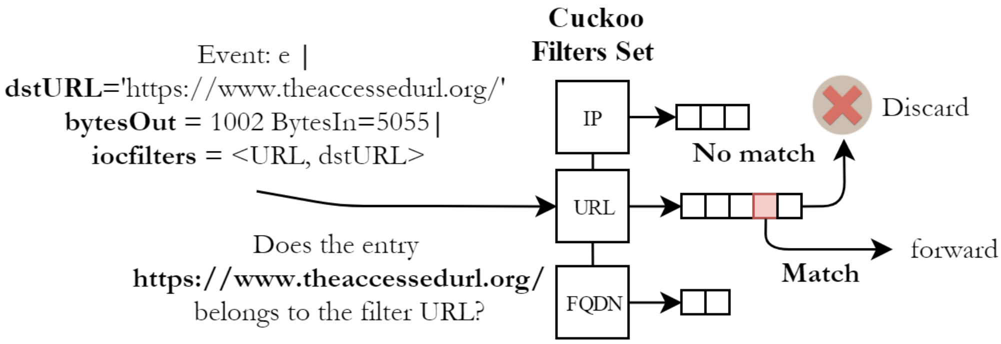


Figure 5.12: Cuckoo Filters - Membership Queries

Using this methodology events can be quickly processed, maintaining an overall low latency. In Chapter 7 we present a comparison of some commonly used data-structures as well as some implementations of probabilistic filters.

Chapter 6

Rules Protection Mechanism

In this chapter, we describe the security measures we took to reduce the potential harm that may come from a rule. Besides the more commonly used security protection mechanisms and protocols, like assuring that the communication between every node is encrypted and guarantying that the clients are authentic, we created a sandbox environment for JavaScript code execution. Allowing users to embed JavaScript code into the rules presents several advantages since they can freely manipulate event fields to create complex detection modules. Although useful, this feature can potentially be used (intentionally or not) to break the system security properties. For instance:

Confidentiality: A malicious user may try to gain access to system resources (e.g. a file) from the JavaScript code.

Integrity: A user may try to change the state of the application if it is accessible by the scripts execution environment.

Availability: A rule can potentially clog the system if it uses more resources than it is supposed to, making it unavailable.

In order to overcome some of the potential vulnerabilities, we created our own sandbox environment using Rhino's [44] API to securely run JavaScript code. In this Chapter, we briefly describe Rhino in Section 6.1 and in Section 6.2 we present the sandbox that we developed using Rhino's API.

6.1 Rhino

Rhino is an open-source implementation of JavaScript developed by Mozilla. Because it was entirely written in Java, it can easily be embedded into Java applications, typically to allow code to be executed at runtime. Importantly for us, Rhino has an API that allows developers to compile and execute JavaScript code according to a set of desired specifications. For instance, the scope (i.e. the set of variables, objects, and functions that the script has access to) of a particular function can be controlled. Using this feature we can restrict the access of a rule's functions from the rest, which is desired because each specific function should be executed in an isolated environment.

The runtime environment used to execute the scripts is called a Context. Each Context must be associated with a specific thread that will be executing the script. The Context holds all the information relative to the scripts execution like the scope and the call stack. The context may also contain custom variables that can be later accessed in the Context Factory.

The Context Factory is one of the most important components of the application. Before creating a context, Rhino will call the Context Factory *makeContext* method that allows the programmer to apply some default security measures to the Context. For instance, the maximum stack depth allowed for a script and the java classes that may be accessed from it. Secondly, it also has a method named *doTopCall* which is called right before a script execution and can be used to associate data to the runtime Context of the script. Additionally, the Context Factory has a method named *observeInstructionCount* that is called every x instructions where x can be defined when creating the Context. Figure 6.1 shows the interaction between the Scope, Context and Context Factory. We omitted some details since they are not relevant to the present work.

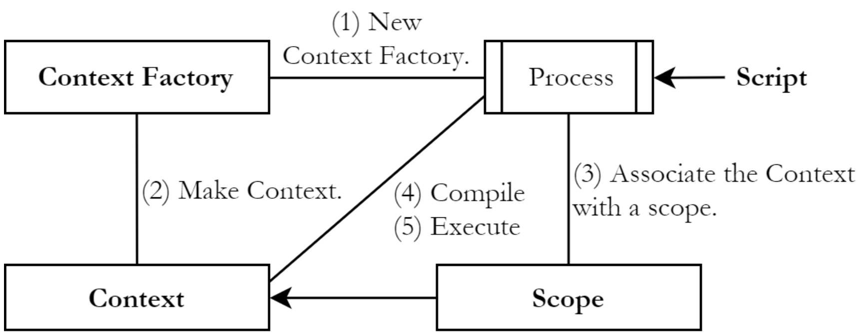


Figure 6.1: Rhino's Execution environment.

By leveraging these features we were able to implement additional protection mechanisms that are described in the next section.

6.2 JavaScript Sandbox

Using Rhino's default protection mechanisms, we began to restrict the scope of each function to only contain our specific libraries. After that, we implemented a specific context that contains our own control variables - the start time, the start memory and a thread id. The start time contains the time at which the script was called for execution, the start memory contains the reserved memory and the thread id contains the identifier of the calling thread. All of the context variables are populated when the script is called for execution in the *doTopCall* method. Additionally, we also added new variables to a custom context factory that can be set by the caller. Namely, the stack depth, the maximum CPU time, the maximum memory and the number of instructions necessary before calling the *observeInstructionCount* method. Having these set of variables in our environment we were able to apply two new specific security measures in the *observeInstructionCount* method. When it is called, we check if either the execution time or if the maximum allowed memory have been exceeded.These two mechanisms assure that if a user incorrectly creates a rule it can be deactivated, both due to large memory allocations or long execution times, which, for instance, may be caused by infinite loops. The remaining variables are simply used in already existing controls. Lastly, we wrapped our custom implementation in an API to make it configurable to the different needs.

```
Sandbox sandbox = new Sandbox()
                    .allowClass(java.util.HashMap.class.getName())
                    .setInstructionsBeforeObserve(5)
                    .setMaxCPUTime(10)
                    .setMaxMemory((int) Math.pow(2, 20))
                    .setStackDepth(3);

sandbox.init();
```

Listing 6.1: JavaSript Sandbox API calls

Listing 6.1 shows an example usage of the API where the only allowed class is the Java HashMap, the resources monitor is called every 5 instructions, scripts have 10 ms to execute, the maximum allowed memory is 1 MB and the stack depth is 3.

This sandbox is used in the nodes of the execution pipeline, which compile and execute JavaScript code. When a new rule arrives at one of these nodes after being parsed it is compiled using the sandbox and stored. When an event is being processed, if it violates some of the protection mechanisms, the sandbox throws an exception, which in turn is caught and used to deactivate that rule.

Chapter 7

Evaluation

In this chapter, we start by comparing the performance of Cuckoo filters against the Bloom filters and other, more commonly used, data structures in Section 7.1. Additionally, we also evaluate the correlation engine in terms of correctness and performance in Section 7.2. To perform the tests we used two machines running CentOS 7 each with 16 GB of RAM and an Intel Xeon E5-2603 1.80GHz CPU . Using one of the machines we evaluated the data structures performance. After that, using both machines, we configured Flink with two task managers, one on each of them. The task managers were configured with 2 task slots, each with 1 GB of reserved memory. Additionally, we also had a job manager running in one of the hosts.

7.1 Probabilistic Filters

In order to evaluate the probabilistic filters, we compare them against commonly used Hash Sets in terms of memory occupancy and search time. The datastructures used in this comparison were the Java default Hash Set implementation and an optimized version of this set - the Trove Hash Set [45]. As for the probabilistic filters, we used a Bloom Filter and a Cuckoo Filter. The Bloom Filter is part of a collection of libraries from Google named Guava [46] and the Cuckoo filter comes from a side project that provides two new filters for Guava [47]. Naturally, the results of the evaluation would differ if we used different implementations. However, we only wanted to establish a baseline in order to understand if the difference in size occupancy and search time was significant enough to justify the use of the probabilistic filters.

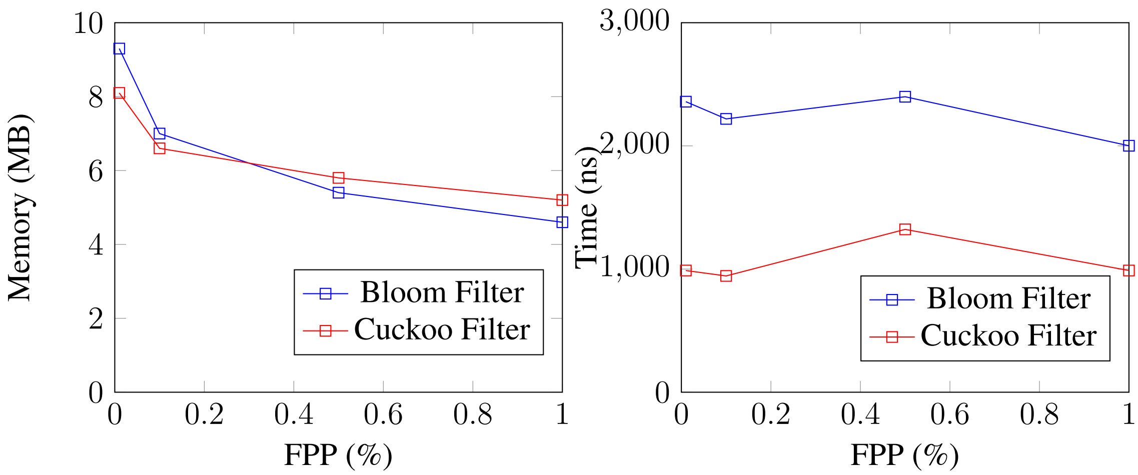
In order to perform the evaluation, we initialized each data structure with the same set of 3 million randomly generated entries of 128 bits. Additionally, we also initialized a set of 300 thousand elements which we use to perform the searches and calculate the average search time. Half of the values belong to the initial set while the other half does not.

For the Trove and Java Hash Sets we simply measure their size occupancy and the average search time. The results of this evaluation are presented in Table 7.1. Since the

Table 7.1: Hash Sets - Object Size and Average Search Time Evaluation

Criteria	Object Size in memory (Mb)	Average Search Time (ns)
Java Hash Set	427	319
Trove Hash Set	353	352

Bloom and Cuckoo filters have the FPP as a parameter, changing this value has an impact both in terms of memory occupancy and search time. Graph 7.1a shows the memory occupancy variations and the Graph 7.1 the search time for four different values of FPP: $1\%, 0.5\%, 0.1\%$ and 0.01%.



(a) Object Size in relation to the FPP (b) Search time in relation to the FPP

Figure 7.1: Bloom and Cuckoo filters evaluation

By looking at the graphs, we can see that the size of the probabilistic filters is several orders of magnitude smaller than the Hash Sets, which shows tremendous optimization. As for the search time, both probabilistic filters are slower. The Bloom filter has to apply several hash functions to each element being searched (to obtain the different array positions), which makes it slower, even in comparison with the Cuckoo filters. The Cuckoo filters, on the other hand, will have to calculate a hash function and a significantly sized fingerprint, although outperforming Bloom filters (less than half the amount of time is needed to find an element in comparison) they are still slower than the Hash sets.

Since we are dealing with small differences of nanoseconds, the search time is negligible even in streaming environments with very low latency requirements, which makes, in our point of view, the trade of advantageous.

7.2 Correlation Engine

This section provides an evaluation of the Correlation Engine. First, in subsection 7.2.1 we evaluate the ability that the engine has to perform accurate computations. Secondly, in subsection 7.2.2 we evaluate the performance of the engine in terms of throughput and latency.

7.2.1 Experimental Evaluation

In order to evaluate the correctness of the correlation engine's rules we created three distinct data source functions: a sawtooth function, a sine wave function and a square wave function [48]. These functions are represented in Figure 7.2.

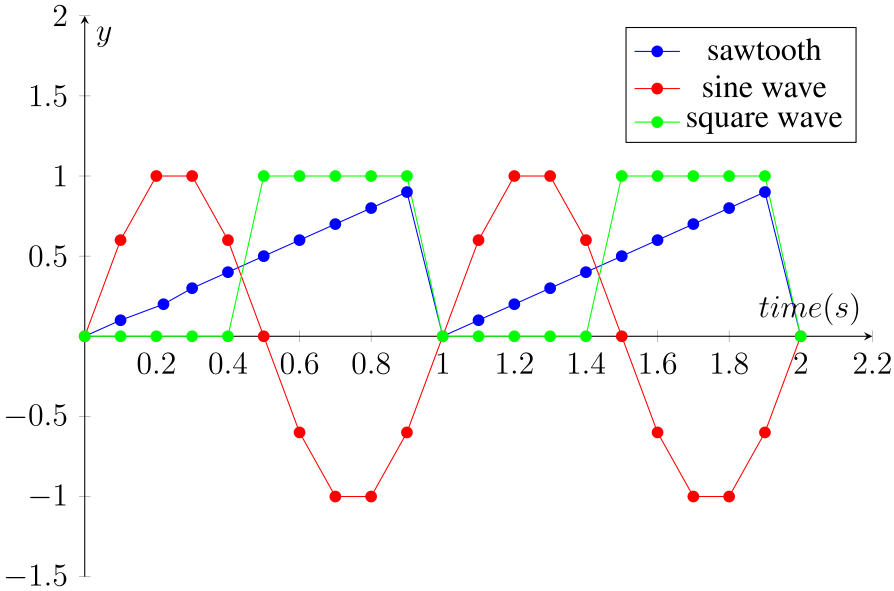


Figure 7.2: Data source functions: saw tooth, sine wave and square wave .

Each of the functions is cyclic. This means that the sum of their values in each cycle, which in this case is one second, is always the same. Therefore, by creating an aggregation rule, using the same interval as the cycles, that outputs the sum of the collected values for each function, we were able to evaluate if the engine was outputting correct results. This allowed an evaluation of the entire pipeline, that is, if thr rule was correctly being created and stored, if the computations were being applied to all of the input events and if the results of the filter and aggregation functions were correct. The rule we used for this evaluation is depicted in the Listing 7.1.

```
RULE "Sum Source Function Data"
  SOURCES source=['sawtooth','sinewave','squarewave']
  USING timestamp
  AGGREGATE BY source
  WITHIN 1:sec,1:sec
  HAVING function (events){
      var result = {'source': events[0].fields['source']};
      var sum = 0;
      for(var i = 0; i < events.length; i++){
```

```
        sum+=events[i].fields['cn1'];
      }
      result['sum'] = sum;
      return result;
    }
  SINK function (event){ return event; }
```

Listing 7.1: Sum source functions data.

By observing the results of the discussed rule, we were, in fact, able to verify that the results of the sawtooth, sine wave and square wave functions were respectively 4.5, 0.0 and 5.0.

7.2.2 Performance

To accurately perform the evaluation of the correlation engine, we created an Event Injector that continuously sends events to Kafka. The events generated by the Injector have seven fields named $a, b, c, d, tag, integer, groupName$ and $groupTag$. The fields a to d represent randomly generated 128 bit values. The field tag contains the string id_tag_x where the $x \in [0, 50] \cap N$. The field $integer$ contains an integer value that also belongs to the same interval as x. The field $groupName$ represents the event source, which can belong to one of six groups. Finally, the $groupTag$ contains the value $"DemoV01"$, which is a simple string that we use to match against a regular expression. The injector sends events to Kafka at a faster rate than the event source node can pull them. We then measure the amount of events that can be processed by the remaining nodes as we keep adding rules to each source group. The rule that we used for the tests is depicted in listing 7.2. The reason we do not include aggregation clauses in the evaluation is that they will vastly vary depending on the aggregation function that will be applied (e.g. average, count or sum) and the size of the window, which adds up to the latency of the execution but it is not related to the execution itself.

```
RULE "G0_id0"
    SOURCES groupName = ['Group0']
    FILTER function filteringDemo(event){
    return utils.matches(event['groupTag'],'([A-Za-z]+) V[0-9]*') &&
                        event['integer']==0 && event['tag']=='id_tag_0';
  }
    IOC FILTERS in id_list(tag)
    SINK function customSink(event){
    return event;
  }
```

Listing 7.2: Evaluation Rule

Each rule has one source that represents the name of the group. In this particular rule, we match events that belong to the source *Group0*. Additionally, the rule filters match the $groupTag$, tag and $integer$ values. These values change as we add new rules to the same source. In this case we are filtering for $integer = 0$ and $tag = id\_tag\_0$. As we add another rule to this group, the values increase by one. Additionally, we also populated a cuckoo filter with a set of a million values, which contains a million distinct id tags. For each six rules that we add to the system (one for each source group), we filter in 2% of the total number of incoming events. The configuration of our job is depicted in Figure 7.3.

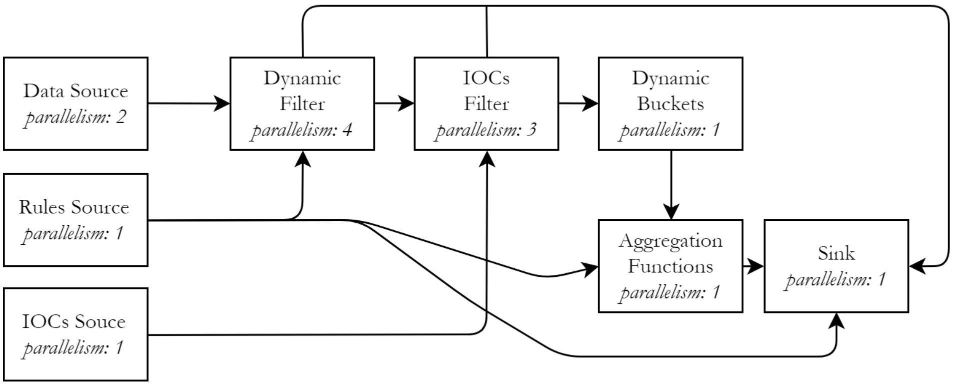


Figure 7.3: Job Evaluation Configuration.

The two most relevant metrics for the evaluation of the correlation engine are the average throughput and latency. To get the throughput we measured the number of events that the engine can process per second as the number of rules increases. Naturally, the throughput will diminish since the computational requirements become more demanding. We start the engine with one rule per source and keep adding rules while measuring the performance. The results of this evaluation are illustrated in Graph 7.4.

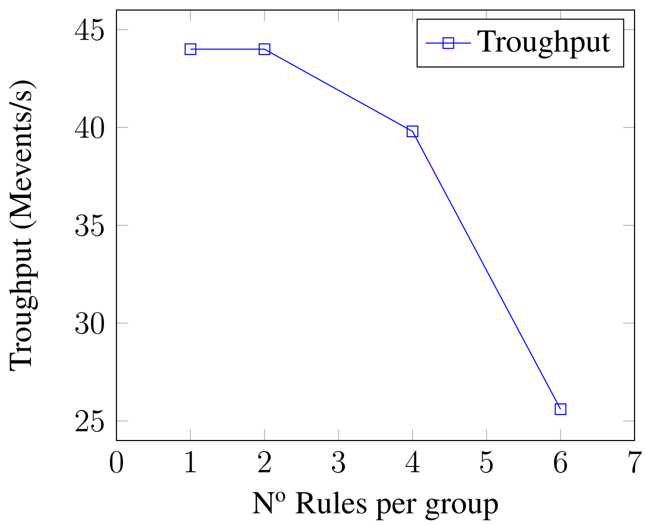


Figure 7.4: Troughput in relation to the number of rules per group

As we can see by the graph, the throughput drops significantly, to about 25 thousand events per second, when the number of rules gets to 6 per group (36 rules in total). This drop in performance is mainly due to two reasons. The first is related to the number of

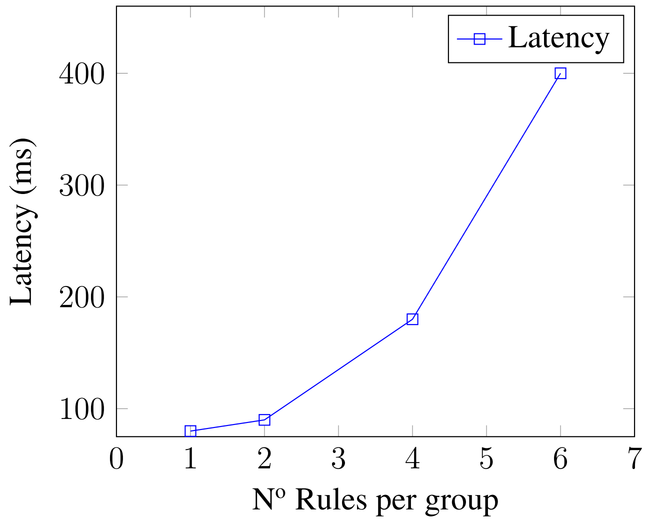


Figure 7.5: Latency in relation to the number of rules per group

iterations that an event has to go through when it enters the filter node. Since each group will have six rules, an event will have to be processed by each one of them. Additionally, the filter processing node was beginning to slow down due to high CPU usage, which could be resolved by adding additional extra computational power.

As for the latency, similarly to the example above, we measured the time that it takes to fully process an event as we keep adding rules. The results are depicted in Graph 7.5. As before, we see a good performance when the number of rules per group is small but as the events have to be processed by a larger number of rules it will take longer for an event to be fully processed before the results can be sent to the proper destinations, hence the larger latency values.

Chapter 8

Conclusions and Future Work

The increasing volume of log data that is produced within an organization and the need to quickly obtain relevant information from that data has brought the necessity of Stream Processing Platforms. In the present work, we presented such a platform that is able to handle a large volume of events and quickly obtain relevant results. Using the developed system, we were able to replicate some of the detection modules that are used in commercial products to detect security threats with the added ability to scale using commodity hardware.

Currently, our system is in a pre-production environment receiving real log data from a small subset of Siemens systems with a volume of about five thousand logs per second. Although our system shows a good performance and the ability to detect security threats, we still have many challenges that we need to overcome. In terms of scaling, we need to improve the way events are matched against a rule in the filter node. In a worst-case scenario, the system may be receiving a large volume of events from a particular source that has a large set of rules, which drastically decreases the performance. To improve the time it takes for an event to match a rule, as future work we may take advantage of Binary Decision Diagrams (BDD) [49], which are used to represent boolean functions as a directed acyclic graph. We may also change our Rhino JavaScript engine to another faster alternative such as Nashorn [50] if we can provide the same security features. Additionally, we can change the filter node to process events asynchronously.

In terms of security, we still need to develop an additional mechanism, such as a sanitizer function, that thoroughly validates users input coming from the rules in order to avoid malicious code injections.

As for the features, we plan to add an enrichment phase that associates new fields to both the incoming events and to the correlated events (i.e. complex events). By associating new data to an event before processing it, users will have more fields at their disposal when creating security rules. Moreover, by adding new fields to the correlated events, such as the geolocation of an IP address, eases the work of security analysts when analyzing the alerts generated by those.

The Cuckoo filters are a major advantage of this system due to their faster response times to a query. However, we still need to develop a better mechanism to populate them. By having both a Cuckoo filter set and a database, the state can easily become inconsistent. For instance, if the communication between the database and the IOC Filter node fails, the changes made to the tables are not translated to the Cuckoo filters creating an inconsistency. Currently, this can only be solved be rescanning the tables and rebuilding the filters.

Lastly, it would be ideal to correlate late events (that arrive beyond the lateness interval) with the persistently stored events (historical data) that belong to the same time frame in order to detect an attack that may have occurred but because of large downtimes of a particular component was not detected.

Additional research and development will also be necessary to further test the Correlation Engine.

Appendix A

Assessment Code

A.1 Esper

```java

public class CEPDemo {


    public static void main (String args []) throws Exception {

    EPServiceProvider epService = EPServiceProviderManager
                                        .getDefaultProvider();
    String expression = "select user,ip, count(*) as total " +
                        "from espertech.Event.win:time(10 sec) " +
                        "where result='denied' " +
                        "group by user,ip " +
                        "having count(*)>=5" ;
    EPStatement statement = epService.getEPAdministrator()
                                          .createEPL(expression);
    statement.addListener(new MyListener());
    //Kafka Consumer
    new ConsumerLoop(0,"Consumer",
                       Arrays.asList("log-stream"),epService).run();
    epService.initialize();
    }
}
```

Listing A.1: Esper Demo Application

```java

public class ConsumerLoop implements Runnable {
  private final KafkaConsumer<String, String> consumer;
  private final List<String> topics;
  private final int id;
  private final EPServiceProvider epService;

  public ConsumerLoop(int id,
                      String groupId,
                      List<String> topics, EPServiceProvider epService)
    {
    this.id = id;
    this.topics = topics;
```

```
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092");
    props.put("group.id", groupId);
    props.put("key.deserializer", StringDeserializer.class.getName());
    props.put("value.deserializer", StringDeserializer.class.getName())
;
    this.consumer = new KafkaConsumer<>(props);
    this.epService = epService;

  }

  @Override
  public void run() {

    try {
      consumer.subscribe(topics);

      while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
          String [] fields = record.value().split(",");
          epService.getEPRuntime().sendEvent(new Event(Integer.parseInt
(fields[0]),fields[1],fields[2],fields[3]));

        }
      }
    } catch (Exception e) {
      // ignore for shutdown
    } finally {
      consumer.close();
    }
  }

  public void shutdown() {
    consumer.wakeup();
  }
}
```

Listing A.2: Kafka Consumer

A.2 Spark

```
public class CEPDemo {

    public static void main(String[] args) throws Exception {
        //Arguments
        if (args.length < 2) {
            System.err.println("Usage: Main <BrokerHost:BrokerPort> <
KafkaTopicsList>");
            System.exit(1);
        }
        CEPDemo engine = new CEPDemo(args[0], args[1]);
        engine.start();
    }

    private JavaStreamingContext jsc;
```

```
    private JavaPairDStream<Tuple3<String, String, String>, Integer> stream;
    private JavaPairDStream<Tuple3<String, String, String>, Integer> alerts;
    private static final int WINDOW_SIZE = 5;

    /**
     *
     * Initialization
     *
     * @param server
     * @param topic
     */
    public CEPDemo(String server, String topic) {
        setupEnvironmnet();
        kafkaConsumer(server, topic);
        rules();
    }

    private void setupEnvironmnet() {
        SparkConf conf = new SparkConf().setAppName("CEPDemo").setMaster("local[2]"); //Two Threads
        jsc = new JavaStreamingContext(conf, Durations.seconds(1));
    }

    /**
     * Kafka Consumer Initialization
     *
     * @param server specifies the adress and port where the kafka server is
     * running
     * @param topic specifies the topic which we want to subscribe
     */
    private void kafkaConsumer(String server, String topic) {
        Map<TopicAndPartition, Long> consumerOffsetsLong = new HashMap<>();
        TopicAndPartition partition = new TopicAndPartition(topic, 0);
        //Kafka Offset specification
        consumerOffsetsLong.put(partition, 0L);
        HashMap<String, String> kafkaParams = new HashMap<>();
        kafkaParams.put("metadata.broker.list", server);
        //Server Configuration. The data will be read as a String and parsed to
        //a tuple of <Event,Integer> where the integer is initially 1 for all events
        stream = KafkaUtils.createDirectStream(jsc,
                String.class, String.class,
                StringDecoder.class, StringDecoder.class,
                String.class, kafkaParams,
                consumerOffsetsLong,
                (MessageAndMetadata<String, String> v1) -> v1.message())
                .mapToPair(new Parser());
    }

    private void rules() {
```

```
        alerts = stream
                //Setup a 5 sec window
                .window(Durations.seconds(WINDOW_SIZE))
                //Accessing the first field of the tuple (_1) and then its action _2()
                .filter((t1) -> t1._1._2().equals("denied"))
                //cont the pair with the same key that is the same (user,ip)
                .reduceByKey((t1, t2) -> {
                    return t1 + t2;
                })
                //if the result is greater or equal to 10, send an alert
                .filter((t1) -> t1._2 >= 10);
    }

    /**
     * Start the execution and print debug information
     *
     * @throws java.lang.InterruptedException
     */
    void start() throws InterruptedException {
        alerts.print();
        jsc.start();                  // Start the computation
        jsc.awaitTermination();    // Wait for the computation to terminate

    }

}
```

Listing A.3: Spark Demo Application

A.3 Flink

```
public class CEPDemo {

  private static final int WINDOW_SIZE = 10;
    public static void main(String[] args) throws Exception {

  //Get user arguments
  if (args.length < 2) {
    System.err.println("Usage: JavaNetworkWordCount <brokerHost:BrokerPort> <kafkaTopics>");
    System.exit(1);
  }
  //Environmet
  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
  //Get input from kafka
  Properties properties = new Properties();
  properties.setProperty("bootstrap.servers", args[0]);
  properties.setProperty("group.id", "Demo");
  DataStream<String> stream = env
    .addSource(new FlinkKafkaConsumer09<>(args[1], new SimpleStringSchema(), properties));
```

```
    //Transformation String -> Event

    DataStream<Event> eventData = stream.map(
      new MapFunction<String,Event>(){
      @Override
      public Event map(String event){
        String[] fields = event.split(",");
          return new Event(Integer.parseInt(fields[0]),fields[1],fields
  [2],fields[3],1);
      }
    }).keyBy("ip","user").timeWindow(Time.of(WINDOW_SIZE, SECONDS)).sum("
  count");

    //Define pattern, denied 5 or more times
    Pattern<Event,?> pattern =
      Pattern.<Event>begin("start")
      .where(new FilterFunction<Event>(){
        @Override
        public boolean filter(Event event){
          return event.result.equals("denied") && event.count >=5;
        }
      });
    PatternStream<Event> patternStream = CEP.pattern(eventData, pattern);
    DataStream<String> warning = patternStream.select(
      new PatternSelectFunction<Event, String>() {
      @Override
      public String select(Map<String, Event> map) throws Exception {
        return "WARN: "+ map.get("start").user + " had its access denied
  5 times to " + map.get("start").ip;
      }
    });

    warning.print();

    //Start Execution
    env.execute("CEPDemo");

  }

}
```

Listing A.4: Flink Demo Application

Appendix B

EPL Grammar

```
/*
 * To change this license header, choose License Headers in Project
    Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

grammar RuleScript;
import ECMAScript;

options{
    tokenVocab = ECMAScript;
}

//---------------
// RULE CLAUSES
//---------------

ruleClause: RULE StringLiteral sourcesClause;
sourcesClause: SOURCES sourcesExpression filterClause; //Continue
filterClause: FILTER filterExpression ( iocFiltersClause | usingClause
    | aggregatebyClause | sinkFunctionClause);
iocFiltersClause: IOC_FILTER iocFiltersExpression (usingClause |
    aggregatebyClause | sinkFunctionClause);
usingClause: USING usingExpression aggregatebyClause;
aggregatebyClause: AGGREGATE_BY aggregateByExpression withinClause;
withinClause: WITHIN withinExpression (sortByClause | havingClause);
sortByClause: SORT_BY sortByExpression havingClause;
havingClause: HAVING havingExpression sinkFunctionClause;
sinkFunctionClause: SINK sinkFunctionExpression;


//SOURCES
sourcesExpression: identifierName Assign OpenBracket StringLiteral (
    Comma StringLiteral)* CloseBracket;

//FILTER
filterExpression: functionDeclaration;

//IN IOC
```

```
iocFiltersExpression: inIocParam*;
inIocParam: (inIoc | notInIoc) OpenParen identifierName CloseParen;
inIoc: IN identifierName;
notInIoc: NOT IN identifierName;

//USING
usingExpression: identifierName;

//AGGREGATE BY
aggregateByExpression: identifierName (Comma identifierName)*;

//WITHIN
withinExpression: numericLiteral Colon TIME_UNIT Comma numericLiteral
    Colon TIME_UNIT ;

//SORT BY
sortByExpression: identifierName;

//HAVING
havingExpression: functionDeclaration;

//SINK FUNCTION
sinkFunctionExpression: functionDeclaration;


//-------------------------------
// LEXER
//-------------------------------

RULE: R U L E;
SOURCES: S O U R C E S;
FILTER: F I L T E R ;
IOC_FILTER: I O C ' ' F I L T E R S;
USING: U S I N G;
AGGREGATE_BY: A G G R E G A T E ' ' B Y;
WITHIN: W I T H I N;
SORT_BY: S O R T ' ' B Y;
HAVING: H A V I N G;
SINK: S I N K;
TIME_UNIT: M S | S E C | M I N;
IN: I N;
NOT: N O T;


fragment A : [aA];
fragment B : [bB];
fragment C : [cC];
fragment D : [dD];
fragment E : [eE];
fragment F : [fF];
fragment G : [gG];
fragment H : [hH];
fragment I : [iI];
fragment J : [jJ];
fragment K : [kK];
fragment L : [lL];
```

```
fragment M : [mM];
fragment N : [nN];
fragment O : [oO];
fragment P : [pP];
fragment Q : [qQ];
fragment R : [rR];
fragment S : [sS];
fragment T : [tT];
fragment U : [uU];
fragment V : [vV];
fragment W : [wW];
fragment X : [xX];
fragment Y : [yY];
fragment Z : [zZ];
```

Glossary

ANTLR Another Tool for Language Recognition.

APT Advanced Presistent Threat.

AVI Atack, Vulnerability and Intrusion.

BDD Binary Decision Diagrams.

BRMS Business Rule Management System.

CEF Common Event Format.

CEP Complex Event Processing.

CPU Central Processing Unit.

CSV Comma Separated Values.

DAG Directed Acyclic Graph.

DFP Dataflow Programming.

EBP Event Batch Processing.

ECMA European Computer Manufacturers Association.

EDA Event-Driven Architectures.

EPL Event Processing Language.

ESP Event Stream Processing.

FPP False Positive Probability.

HIDS Host Intrusion Detection System.

IDS Intrusion Detection System.

IOC Indicator of Compromise.

IPS Intrusion Prevision System.

JVM Java Virtual Machine.

NIDS Network Intrusion Detection System.

POC Proof of concept.

POJO Plain Old Java Object.

RAM Random Access Memory.

RDD Resilient Distributed Data-sets.

SPP Stream Processing Platform.

Bibliography

[1] S. Allen, "Importance of understanding logs from an information security standpoint," tech. rep., SANS, 2001. [Online; accessed 01-Nov-2016].

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST)*, IEEE, June 2010.

[3] M. Divya and S. Goyal, "Elasticsearch an advanced and quick search technique to handle voluminous data," *COMPUSOFT : International Journal of Advanced Computer Technology*, 2013.

[4] Apache, "Hydra." https://github.com/addthis/hydra, 2016. [Online; accessed 28-nov-2016].

[5] K. Banker, *MongoDB in Action*. Manning Publications, 2011.

[6] K. M. Candy, *Event Driven Architecture*, pp. 1040–1044. Boston, MA: Springer US, 2009.

[7] "Syslog rfc." https://www.ietf.org/rfc/rfc3164.txt. [Online; accessed 12-Jun-2017].

[8] "Cef." https://community.saas.hpe.com/t5/ArcSight-Connectors/ArcSight-Common-Event-Format-CEF-Guide/ta-p/1589306?attachment-id=61491. [Online; accessed 12-Jun-2017].

[9] T. Akidau, "The world beyond batch: Streaming 101." [Online; accessed 24-nov-2016], 2015 2015.

[10] R. R. Mohd. Saboor, "Designing and developing complex event processing applications," tech. rep., Sapient, 2013. [Online; accessed 08-Dez-2017].

[11] H. Dominique, "Complex event processing (cep) primer," tech. rep., 2001. [Online; accessed 08-Dez-2017].

[12] M. R. N. Mendes, P. Bizarro, and P. Marques, *A Performance Study of Event Processing Systems*, pp. 221–236. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[13] T. B. Sousa, “Dataflow programming concept, languages and applications,” 2012.

[14] N. Marz, *Big data : principles and best practices of scalable realtime data systems*. [S.l.]: O’Reilly Media, 2013.

[15] J. Kreps, “Questioning the lambda architecture.” https://www.oreilly.com/ideas/questioning-the-lambda-architecture, 2014. [Online; accessed 18-out-2016].

[16] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, pp. 422–426, July 1970.

[17] T. D. Hodes, S. E. Czerwinski, B. Y. Zhao, A. D. Joseph, and R. H. Katz, “An architecture for secure wide-area service discovery,” *Wireless Networks*, vol. 8, no. 2, pp. 213–230, 2002.

[18] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, “Cuckoo filter: Practically better than bloom,” in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’14, (New York, NY, USA), pp. 75–88, ACM, 2014.

[19] R. Pagh and F. F. Rodler, “Cuckoo hashing,” *J. Algorithms*, vol. 51, pp. 122–144, May 2004.

[20] P. Verissimo and L. Rodrigues, *Distributed Systems for System Architects*. Norwell, MA, USA: Kluwer Academic Publishers, 2001.

[21] “Espertech - event series intelligence.” http://www.espertech.com/esper/. [Online; accessed 28-nov-2016].

[22] “Drools.” https://www.drools.org/. [Online; accessed 28-nov-2016].

[23] “Wso2 complex event processor.” http://wso2.com/products/complex-event-processor/. [Online; accessed 28-nov-2016].

[24] “Apache spark lightning-fat cluster computing.” http://spark.apache.org/. [Online; accessed 28-nov-2016].

[25] “Apache storm.” http://storm.apache.org/. [Online; accessed 28-nov-2016].

[26] “Apache flink - open source platform for distributed stream and batch data processing.” https://flink.apache.org/. [Online; accessed 28-nov-2016].

[27] “Apache hadoop yarn.” https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html. [Online; accessed 18-Dez-2016].

[28] T. Dudziak's, "Storm and esper." https://tomdzk.wordpress.com/2011/09/28/storm-esper/, 2011. [Online; accessed 18-out-2016].

[29] "Siddhi." https://github.com/wso2/siddhi. [Online; accessed 18-Dez-2016].

[30] Y. Wang, "Stream processing systems benchmark: Streambench," Master's thesis, Aalto University, School of Science, 2016-06-13.

[31] R. Shukla, P. Pandey, and V. Kumar, "Big data frameworks: At a glance," *International Journal of Innovations and Advancement in Computer Science*, vol. 4, January 2015.

[32] Y. Zhou, "A study on implementing iterative algorithms using bigdata frameworks." [Online; accessed 28-nov-2016].

[33] J. Kreps, N. Neha, and R. Jun, "Kafka: A distributed messaging system for log processing," in *In Proceedings of the NetDB*, pp. 1–7, 2011.

[34] "Esperha." http://www.espertech.com/products/esperha.php. [Online; accessed 28-nov-2016].

[35] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010.

[36] "Akka - build powerful concurrent and distributed applications mode easily." http://akka.io/. [Online; accessed 09-jan-2017].

[37] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[38] "Netbeansplatform." https://netbeans.org/features/platform/. [Online; accessed 12-Jun-2017].

[39] "Bitbucket." https://bitbucket.org/. [Online; accessed 12-Jun-2017].

[40] "Antlr (another tool for language recognition)." https://github.com/antlr/antlr4. [Online; accessed 24-fev-2017].

[41] F. Gyula and A. Mattias, "Rbea: Scalable real-time analytics at king." http://data-artisans.com/rbea-scalable-real-time-analytics-at-king/, 2016. [Online; accessed 01-Nov-2016].

[42] "Groovy." http://www.groovy-lang.org/documentation.htmli. [Online; accessed 23-Dez-2016].

[43] "Ecma script." https://github.com/antlr/grammars-v4/tree/master/ecmascript. [Online; accessed 15-Mar-2017].

[44] "Rhino." https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino. [Online; accessed 24-fev-2017].

[45] "Trove high performance collections for java." http://trove.starlight-systems.com/overview. [Online; accessed 10-jan-2017].

[46] "Bloom filter." https://github.com/google/guava/blob/master/guava/src/com/google/common/hash/Blo [Online; accessed 10-jan-2017].

[47] "Guava-probably: Probabilistic filters." https://github.com/bdupras/guava-probably. [Online; accessed 10-jan-2017].

[48] "Bitbucket." https://github.com/dataArtisans/oscon. [Online; accessed 20-Feb-2017].

[49] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, pp. 677–691, Aug. 1986.

[50] "Nashorn." http://openjdk.java.net/projects/nashorn/. [Online; accessed 24-fev-2017].